Hein Meling

Adaptive Middleware Support and Autonomous Fault Treatment: Architectural Design, Prototyping and Experimental Evaluation

Doctoral thesis
for the degree of doktor ingeniør

Trondheim, May 2006

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics
and Electrical Engineering
Department of Telematics

NTNU
Innovation and Creativity

NTNU
Norwegian University of Science and Technology

Doctoral thesis
for the degree of doktor ingeniør

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Telematics

ISBN 82-471-7938-5 (printed version)
ISBN 82-471-7937-7 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2006:92

Printed by NTNU-trykk

Dedicated to Håvard and Elin

Abstract

Networked computer systems are prevalent in most aspects of modern society, and we have become dependent on such computer systems to perform many critical tasks. Moreover, making such systems *dependable* is an important goal. However, dependability issues are often neglected when developing systems due to the complexities of the techniques involved.

A common technique used to improve the dependability characteristics of systems is to replicate critical system components whereby the functions they perform are repeated by multiple replicas. Replicas are often distributed geographically and connected through a network as a means to render the failure of one replica independent of the others. However, the network is also a potential source of failures, as nodes can become temporarily disconnected from each other, introducing an array of new problems.

The majority of previous projects have focused on the provision of middleware libraries aimed at simplifying the development of dependable distributed systems, whereas the pivotal deployment and operational aspects of such systems have received very little attention. This thesis extends on previous works and emphasize the deployment and operational aspects, where the gain in terms of improved dependability is likely to be the greatest.

The main contribution of this dissertation is an architecture for autonomous replication management, aimed to improve the dependability characteristics of systems through a self-managed fault treatment mechanism that is adaptive to network dynamics and changing requirements. Consequently, the architecture also improves the deployment and operational aspect of systems, and reduces the human interactions needed. The architecture has been implemented as a proof of concept prototype by extending the Jgroup object group system.

In addition, numerous supporting contributions are also included in this work: (i) an architecture for dynamic protocol composition that avoids the delays of event processing in intermediate layers of a strictly vertical protocol stack; (ii) adaptive protocol selection is also made possible on a per method/invocation basis, by annotating server methods with the replication protocol to be used; (iii) client-side membership handling is also implemented aimed to improve the load balancing and failover properties of systems when exposed to failures; (iv) online upgrade management of operational services is also implemented as an extension to the replication management architecture.

Finally, the dissertation provides extensive experimental evaluation of the fault treatment capabilities of the autonomous replication management architecture, with emphasis on testing complex failure scenarios. The first experiment examines the ability of clients to maintain correct membership when servers crash and recover. The second experiment investigates the behavior of services when exposed to multiple nearly-coincident node crash failures. In conjunction with this experiment, a novel technique has been developed to estimate various service dependability characteristics. In the third experiment the recovery performance of a system deployed in a wide area network is evaluated. In this experiment multiple nearly-coincident reachability changes are injected to simulate network partitions separating the service replicas.

To support the experimental evaluation, a set of generic tools have also been developed to aid the execution and analysis of the experiments.

Preface

The Portable Document Format (PDF) version of this thesis features hyperlinks, enabling the reader to click on chapter, section and figure references, citations and other linked items in order to navigate easily within the document. Also, the reader is encouraged to take note of the back reference links used in the bibliography; these enable you to move to the pages where a particular citation was made.

The Jgroup/ARM [81] dependable computing toolkit presented in this thesis is made available as open-source, licensed under the GNU Lesser General Public License (LGPL). It can be downloaded from http://jgroup.sourceforge.net/. The Jgroup Object Group System [87] was originally developed at the University of Bologna, Italy, by Alberto Montresor under the supervision of Professor Özalp Babaoğlu.

Acknowledgements

First of all, I wish to thank my thesis advisor Professor Bjarne E. Helvik for his contributions to this work, both as a coauthor of several papers and for invaluable guidance and encouragement which eventually lead to the completion of this work. I also wish to thank my mentor, Associate Professor Alberto Montresor for his contributions and technical assistance concerning the Jgroup Object Group System. Also for all the administrative help and your kindness during my visits to the University of Bologna and the beautiful city of Verona. Also thanks to both Alberto and Professor Özalp Babaoğlu for the collaboration on the Jgroup and Anthill projects.

I would also like to thank Associate Professor Simin Nadjm-Tehrani, Dr. Oddvar Risnes and Associate Professor Poul E. Heegaard for taking the time to serve on my dissertation committee. I'm also very grateful to Ketil Kristiansen for proofreading the dissertation.

Thanks to all my colleagues at the Department of Electrical and Computer Engineering (IED) at the University of Stavanger (UiS) for contributing to a joyful and friendly work atmosphere. In particular, I wish to thank Professor Sven Ole Aase and Kjell Olav Kaland for reducing my workload this last semester. Thanks to my former colleagues at the Department of Telematics (ITEM) at the Norwegian University of Science and Technology (NTNU), especially Arne Øslebø, Otto Wittner, Tønnes Brekne, Jacqueline Floch and Frank Li.

Thanks to Patricia Retamal (IED) and Randi Fløsnes (ITEM) for all kinds of administrative help, and Pål Sturla Sæther and Asbjørn Karstensen (ITEM), Birger Sandvik and Theodor Ivesdal (IED) for invaluable technical assistance. Also thanks to the students who have contributed code to the Jgroup/ARM toolkit: Rohnny Moland, Tor Arve Stangeland, Rune Vestvik, Henning Hommeland and Jo Andreas Lind.

Finally, and most of all thanks to my wife Ingrid for her loving support through all the ups and downs and for making me finish what I started even though it has taken way too long to do it.

Publications by the Author

Published parts of this thesis

[1] Hein Meling, Alberto Montresor, Bjarne E. Helvik, and Özalp Babaoğlu. Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management. Technical Report No. 11, University of Stavanger, January 2006. Submitted for publication.

[2] Bjarne E. Helvik, Hein Meling, and Alberto Montresor. An Approach to Experimentally Obtain Service Dependability Characteristics of the Jgroup/ARM System. In *Proceedings of the Fifth European Dependable Computing Conference (EDCC)*, Lecture Notes in Computer Science, pages 179–198. Springer-Verlag, April 2005.

[3] Hein Meling and Bjarne E. Helvik. Performance Consequences of Inconsistent Client-side Membership Information in the Open Group Model. In *Proceedings of the 23rd International Performance, Computing, and Communications Conference (IPCCC)*, Phoenix, Arizona, April 2004.

[4] Hein Meling, Jo Andreas Lind, and Henning Hommeland. Maintaining Binding Freshness in the Jgroup Dependable Naming Service. In *Proceedings of Norsk Informatikkonferanse (NIK)*, Oslo, Norway, November 2003.

[5] Marcin Solarski and Hein Meling. Towards Upgrading Actively Replicated Servers on-the-fly. In *Proceedings of the Workshop on Dependable On-line Upgrading of Distributed Systems in conjunction with COMPSAC 2002*, Oxford, England, August 2002.

[6] Hein Meling and Bjarne E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS)*, Bertinoro, Italy, May 2001.

Other publications

[1] Alberto Montresor, Hein Meling, and Özalp Babaoğlu. *Toward Self-Organizing, Self-Repairing and Resilient Distributed Systems*, chapter 22, pages 119–124. Number 2584 in Lecture Notes in Computer Science. Springer-Verlag, June 2003.

[2] Özalp Babaoğlu, Hein Meling, and Alberto Montresor. Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[3] Alberto Montresor, Hein Meling, and Özalp Babaoğlu. Messor: Load-Balancing through a Swarm of Autonomous Agents. In *Proceedings of the International Workshop on Agents and Peer-to-Peer Computing in conjunction with AAMAS 2002*, Bologna, Italy, July 2002.

[4] Alberto Montresor, Hein Meling, and Özalp Babaoğlu. Towards Self-Organizing, Self-Repairing and Resilient Large-Scale Distributed Systems. In *Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo)*, Bertinoro, Italy, June 2002.

[5] Alberto Montresor, Hein Meling, and Özalp Babaoğlu. Towards Adaptive, Resilient and Self-Organizing Peer-to-Peer Systems. In *Proceedings of the International Workshop on Peer-to-Peer Computing (co-located with Networking 2002)*, Pisa, Italy, May 2002.

[6] Hein Meling, Alberto Montresor, and Özalp Babaoğlu. Peer-to-Peer Document Sharing using the Ant Paradigm. In *Proceedings of Norsk Informatikkonferanse (NIK)*, Tromsø, Norway, November 2001.

[7] Finn Arve Aagesen, Bjarne E. Helvik, Ulrik Johansen, and Hein Meling. Plug and Play for Telecommunication Functionality – Architecture and Demonstration Issues. In *Proceedings of the International Conference on Information Technology for the New Millennium (IConIT)*, Bangkok, Thailand, May 2001.

[8] Finn Arve Aagesen, Bjarne E. Helvik, Vilas Wuwongse, Hein Meling, Rolv Bræk, and Ulrik Johansen. Towards a Plug and Play Architecture for Telecommunications. In Thongchai Yongchareon, Finn Arve Aagesen, and Vilas Wuwongse, editors, *Proceedings of the IFIP TC6 Fifth International Conference on Intelligence in Networks (SmartNet)*, pages 321–334, Pathumthani, Thailand, November 1999. Kluwer Academic Publishers.

[9] Finn Arve Aagesen, Bjarne E. Helvik, Hein Meling, and Ulrik Johansen. Plug and Play for Telecommunications – Architecture and Demonstration Issues. In *Proceedings of Norsk Informatikkonferanse (NIK)*, Trondheim, Norway, November 1999.

[10] Audun Jøsang, Hein Meling, and May Lu. The establishment of public key infrastructures; Are we on the right path? In *Proceedings of Norsk Informatikkonferanse (NIK)*, Trondheim, Norway, November 1999.

Technical reports

[1] Hein Meling, Alberto Montresor, Özalp Babaoğlu, and Bjarne E. Helvik. Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing. Technical Report UBLCS-2002-12, Department of Computer Science, University of Bologna, October 2002.

[2] Alberto Montresor and Hein Meling. Jgroup Tutorial and Programmer's Manual. Technical Report UBLCS-2000-13, Department of Computer Science, University of Bologna, September 2000. Revised February 2002.

[3] Hein Meling and Bjarne E. Helvik. Dynamic Replication Management; Algorithm Specification. Plug-and-Play Technical Report 1/2000, Department of Telematics, Trondheim, Norway, December 2000.

[4] Hein Meling and Bjarne E. Helvik. Dynamic Replication Management; Implementation Options. Plug-and-Play Technical Report 2/2000, Department of Telematics, Trondheim, Norway, December 2000.

[5] Bjarne E. Helvik and Hein Meling. Dynamic Replication; A simple dependability model and some considerations concerning optimal state synchrony between active and passive replicas. Plug-and-Play Technical Report 3/2000, Department of Telematics, Trondheim, Norway, December 2000.

[6] Ulrik Johansen, Finn Arve Aagesen, Bjarne E. Helvik, and Hein Meling. Demonstrator – Requirements and Functional Description. Plug-and-Play Technical Report 3/1999, Department of Telematics, Trondheim, Norway, 1999.

[7] Finn Arve Aagesen, Rolv Bræk, Jacqueline Floch, Bjarne E. Helvik, Ulrik Johansen, Hein Meling, and Vilas Wuwongse. A Reference Model for Plug and Play. Plug-and-Play Technical Report 1/1999, Department of Telematics, Trondheim, Norway, 1999.

Contents

Nomenclature

BW	the bandwidth used by the kernel density function
c_k	command k associated with a service interface
d_x	number of Jgroup daemons in site x
D_i	the duration to reach a stable state after injection I_i
$D\sharp$	state identifier (down state)
δ_s	the setup delay for the injection of a network partition
δ_c	the commit delay for the injection of a network partition
$e_{i,j}$	event j associated with the i^{th} listener interface
E_i	last event system event before injection I_{i+1}
$\mathfrak{F}$	the set of down states
f	number of failures to be tolerated
f_{i_l}	time of the l^{th} failure, relative to the first failure in the i^{th} trajectory
$\mathcal{G}$	the set of group members (replicas) in a group
$\mathcal{G}^n$	the set of group members (replicas) in group n
$\mathcal{G}_i^n$	the set of group members (replicas) in group n at time i
$g(\underline{X}_i)$	generic function on samples from $\underline{X}_i$
I_i	injection number i
$I(\cdots)$	indicator function
i_j	the j^{th} event in the i^{th} failure trajectory
k	number of fault injections
k^*	the reached stratum (the actual number of faults injected)
l	index counter for the number of failures injected so far
$\hat{\Lambda}$	system failure intensity
λ	node failure rate

m	a message
m_i	message number i
N	number of observations
N_x	number of nodes in site x
Ni	node number i
n	number of nodes in target system
P_i	the set of patterns from which injection number i is drawn
$\mathcal{P}$	the current set of partitions
p_i	probability of failure trajectory i
$p_{j,i}$	the i^{th} reachability pattern to inject in the j^{th} experiment
π_k	probability of a trajectory in stratum $\mathcal{S}_k$
R	the current redundancy level of a group (or service)
R_p	the current redundancy level of a group (or service) in partition p
R_{min}	the minimum redundancy level of a group (or service)
$R_{\text{min}}\ (s)$	the minimum redundancy level of service s
R_{init}	the initial redundancy level of a group (or service)
R_{active}	the number of active replicas
R_{passive}	the number of passive replicas
$\hat{R}(t)$	the predicted reliability function
S_i	server number i
$S_A(i)$	server number i of type A
$\mathcal{S}_k$	classification/stratum for trajectories with k concurrent failure events
σ_k	variance in the duration of a trajectory reaching stratum $\mathcal{S}_k$
$(T\|k = l)$	the duration of a trajectory that completes in stratum $\mathcal{S}_l$
T_{max}	maximum range from which injections are chosen
T_{min}	minimum time between injections
T_{BF}	time between failures
T_{NR}	node recovery time
T_{SR}	service recovery time
$T_{\|\mathcal{V}\|}$	renewal rate for replica monitoring given a group size of $\|\mathcal{V}\|$
T_{cu}	client update time
T_f	client-side failover latency
T_i	duration of trajectory i

T_r	server recovery time
T_u	total client-side update delay
t_d	client-side detection time for server failure
t_{i_j}	time of the j^{th} event in the i^{th} failure trajectory
t_u	time at which the client-side proxy is updated again
t_x	failure occurrence time
Θ	expected duration of a trajectory
Θ_k	expected duration of a trajectory reaching stratum $\mathcal{S}_k$
$\hat{U}$	service unavailability
$\hat{v}$	version number of the replica
v_k	view number k (view identifier)
$\mathcal{V}$	the set of members in the current view
$\mathcal{V}_k$	the set of members in the view identified by v_k
$\mathcal{V}_p$	the set of members in the current view in partition p
$\|\mathcal{V}\|$	the cardinality of view $\mathcal{V}$
View-c	a view event with cardinality c
$X\sharp$	state identifier (up state)
X_{i_j}	state after event i_j
$X_i(t)$	the state at time t in the i^{th} failure trajectory
$X_i(t_x) \bowtie \mathsf{f}$	a trajectory i where a failure occurs at t_x
$\underline{X}_i$	list of events and timestamps recorded for trajectory i
Y_i	generic function on samples from $\underline{X}_i$
Y_i^d	time spent in a down state during trajectory i
Y_i^f	indicator for visiting down state during trajectory i

Abbreviations

AOP	Aspect-Oriented Programming
API	Application Programming Interface
ARM	Autonomous Replication Management
AS	Additional Service
CORBA	Common Object Request Broker Architecture
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DR	Dependable Registry
EGMI	External Group Method Invocation
EGMIS	External Group Method Invocation Service
EJB	Enterprise Java Beans
GC	Garbage Collection
GCS	Group Communication System
GM	Group Manager
GMI	Group Method Invocation
GMIS	Group Method Invocation Service
GMS	Group Membership Service
IGMI	Internal Group Method Invocation
IGMIS	Internal Group Method Invocation Service
IIOP	Internet-Inter ORB Protocol
IP	Internet Protocol
ISO	International Organization for Standardization
JD	Jgroup Daemon
JVM	Java Virtual Machine

LAN	Local Area Network
LMI	Local Method Invocation
MANET	Mobile Ad Hoc Network
MC	Management Client
MDT	Mean Down Time
MS	Monitored Service
MTBF	Mean Time to Between Failures
NTP	Network Time Protocol
OD	Outdated view
OGS	Object Group System
ORB	Object Request Broker
OSI	Open Systems Interconnection
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PE	Partition Emulator
PGMS	Partitionable Group Membership Service
RM	Replication Manager
RMI	Remote Method Invocation
RMS	Reliable Multicast Service
ROWA	Read-One, Write-All
SM	Service Monitor
SMS	State Merging Service
TCP	Transmission Control Protocol
TINA	Telecommunications Information Networking Architecture
UDP	User Datagram Protocol
WAN	Wide Area Network
XML	eXtensible Markup Language

Part I

Overview of Research

Chapter 1

Introduction

The increasing reliance on networked information systems in modern society requires that the services they provide remain *available* and the actions they perform be *correct*. A common technique for achieving these goals is to *replicate* critical system components whereby the functions they perform are repeated by multiple replicas. As long as replica failures are independent, the technique provides higher availability and correctness for the system than that of its individual components. Distributing replicas geographically and connecting them through a network is often effective for rendering failures independent. However, the network is in no way static. Nodes fail, are removed or introduced, and temporary network partitions may occur. Providing highly dependable services cost-efficiently in such an environment, requires self-managing systems capable of fault treatment in response to failures.

Traditionally, vendor specific hardware based solutions have been used to provide dependable computing. In the last decade the trend has been towards using replicated *commercial-off-the-shelf* (COTS) components, augmented with specialized software components that enable dependable computing. The rationale behind this trend is mainly driven by the fact that COTS components are cheaper, but they are also more flexible and easier to replace. In addition, the evolution of processing capacity of such components are typically superior to custom made hardware, due to the lower production volume of such hardware.

In parallel with the trend towards replicating COTS components, distributed computing has gained an enormous growth due to the commercialization of the Internet. This growth, and the complexity of the systems involved, has lead to the emergence of numerous distributed computing platforms, often called *middleware* because they

appear between the application and operating system services. These middleware platforms greatly simplify the development of distributed software applications, since they provide high-level programming interfaces for building distributed applications, thus hiding the low-level details, such as remote communication and object location. Most notable are CORBA [50] and Java RMI [123], and more recently Java 2 Enterprise Edition [120], Jini [7] and .NET [39], all of which hold the promise of simplifying network application complexity and development efforts. Their ability to exploit COTS components, cope with heterogeneity and distribution and to permit access to legacy systems makes them particularly attractive for building application servers and three-tier e-business solutions.

Facilitating simplicity in distributed application development, deployment and operation, middleware frameworks typically implement a number of transparencies [125]. Collectively these transparencies are often referred to as the distribution transparency, and include *location transparency* and *access transparency* among others. Unfortunately, most middleware frameworks do not provide *replication transparency* and *failure transparency*, facilitating ease of development and operation of dependable distributed applications. This shortcoming has been recognized by numerous academic research projects [43, 87, 18, 93, 96], and also by the Object Management Group (the governing body for the CORBA standard) in its Fault Tolerant CORBA specification [49]. The telecommunications community have always had strong focus on dependable service delivery, and hence the TINA[1] Consortium (see [128]) emphasized dependability in their various specifications, e.g. [129, 130]. The TINA architecture reuse CORBA as its distributed processing environment, with a few telecommunications related enhancements. In recent years, commercial [10, 117] implementations of dependable middleware products have also become available.

The majority of previous projects have focused on the provision of middleware libraries aimed at simplifying development of dependable distributed applications, whereas the pivotal deployment and operational aspects of such applications have received very little attention. This thesis extends on previous works and emphasize the deployment and operational aspects, where the gain in terms of improved dependability is likely to be the greatest.

This thesis focus on *object-oriented* middleware frameworks based on the *client-server* paradigm, where a client object issues requests to a server object that performs the operation associated with the request, and returns an appropriate response to the client.

[1] Telecommunications Information Networking Architecture

1.1 Motivations

Replication transparency is about hiding the fact that an object is replicated [125], and requires that the client be able to communicate with the replicated server as if it was a single entity. The lack of support for replication transparency in existing middleware environments stem from their fundamental one-to-one interaction model. Such environments would have to simulate a "one-to-many" interaction model through multiple one-to-one interactions [48]. This approach not only increases application complexity, since the application developer needs to worry about complex reliability protocols, but it also degrades performance.

A common approach to enhance a middleware platform with replication transparency is to introduce replicated (server) objects. Hence, the unit of replication is an object. Thus, for a client object to communicate transparently (and efficiently) with a replicated server object, it needs a one-to-many communication primitive. Such one-to-many interactions can be provided by a *group communication system* [87, 29]. Group communication provides support for managing groups of objects and primitives for communication between all members of a group. The purpose of a group is to provide a single logical address, through which clients can transparently communicate with an object group, as if it were a single, non-replicated object. Clients can communicate with an object group, without knowing the location and identity of the individual members. The clients does not need to know the size of the group. The notion of object groups has shown itself to be an important paradigm for building applications that support fault tolerance, high availability, load balancing, and parallel processing.

Although there are many middleware platforms based on object groups [43, 87, 18, 3] that support replication transparency, most platforms do not support failure transparency. Failure transparency hides, from a client, the failure and recovery of a server object. Failure transparency is an important aspect in the design of (possibly unmaintained) highly available systems. Traditionally, group communication systems assume a dynamic crash no-recovery model [134]. That is, it will hide the fact that a server has failed, using the group mechanism for hiding the number of members of the group, their location and identity. However, a group communication system per se will not try to recover a failed member. Maintaining a certain redundancy level (and thereby service availability) requires that replica failures are detected and handled through manual intervention, or through some unspecified external entity [134] [29, and references therein]. Under this assumption, one can only hide failure until all initial group members are exhausted, which will eventually happen unless there is active maintenance of the system.

The overall goal of this thesis is to propose a *self-managed fault treatment* middleware architecture, consequently improving the dependability characteristics of services deployed on top of the architecture.

1.2 About this Thesis

This thesis was started in the context of the Plug-and-Play (PaP)[2] project [126] – a joint project between the Norwegian University of Science and Technology (NTNU) and SINTEF. The PaP project aims at developing an architecture for network-based service systems with A) flexibility and adaptability, B) robustness and survivability, and C) QoS awareness and resource control. The goal is to enhance the flexibility, efficiency and simplicity of system installation, deployment, operation, management and maintenance by enabling dynamic configuration of network components and network-based service functionality.

The work in this thesis was triggered by the intention to provide dependable PaP services [1] through middleware functionality which a) were transparent to the service functionality, b) had a replication level (and style) tailored to the dependability requirements of the individual services (objects or actors in PaP terminology) and c) were autonomously maintained under node failures and topology changes in the network, etc.

Since the inception of the PaP project, we have seen a significant interest in systems capable of coping with complexity and dynamism in the same spirit as we started out with in the PaP project. For instance, the *Autonomic Computing* initiative proposed by IBM in March 2001 [9] and the *BISON* project [24]. Since then, the *Autonomic Communication* concept has evolved [8] based on the same ideas, but instead focusing on autonomy in network environments, much like the overall ideas of the PaP project.

An autonomic computing system tries to mimic the human autonomous nervous system, which takes care of the actions you have no conscious control over, such as breathing, heartbeat, digestion and so on [95]. An autonomic computing system is therefore considered to be *self-managing*. The properties of a self-managing system may include: *self-configuring, self-healing, self-optimizing* and *self-protecting*. Such capabilities can be obtain through an *engineered approach* or through an *emergent*

[2]The Plug-and-Play project has later been dubbed TAPAS (Telematics Architecture for Play-based Adaptable Systems).

behavior based approach. Emergent behavior is the overall behavior generated by many simple behaviors interacting in some way, where a simple behavior is a behavior with no true awareness of the overall emergent behavior it is part of [135]. Emergent behaviors are common in nature and have been observered in colonies of social insects and animals. The emergent behavior based approach is very appealing when the sheer size of the system makes traditional engineered techniques infeasible. For instance, emergent behavior based approaches have been used to construct load balancing mechanisms [90], a P2P file-sharing system [16], and to resolve network management issues [135], all of which focus on complex and large-scale systems.

The work presented in this thesis bring along many of the ideas developed in the PaP project, and draws on concepts from autonomic computing and communication to develop a self-managing fault treatment architecture, based on a traditional engineered approach.

1.2.1 Research Objectives and Constraints

A lot of research has been done in the domain of distributed computing and group communication in recent years, and very promising techniques and platforms have been proposed to deal with the various transparencies that are so sought after to reduce application complexity. Yet very few proposals [102, 103, 81] have focused on the *fault treatment* issue.

The overall goal of this thesis is to provide a fault-tolerance architecture that is self-managing and adaptive to network dynamics and changing requirements. The added benefits of such an architecture are twofold:

- reduced human interactions and costs, and
- improved dependability of systems using the architecture [57, 81].

To reach this goal, an architecture for *Autonomous Replication Management* (ARM) is proposed. ARM extends the Jgroup [87] object group system, in a manner which allows the deployment and operation of services through an *autonomic management* facility, consequently reducing the required human interactions and costs. Another important goal of this thesis is to evaluate the dependability characteristics obtained by using the ARM architecture [57].

The assumed *target environment* for such a system is one in which the distributed system contains a pool of nodes (processors) on which service replicas can be hosted (see Figure 1.1). It is also assumed that more than one service can be hosted on the

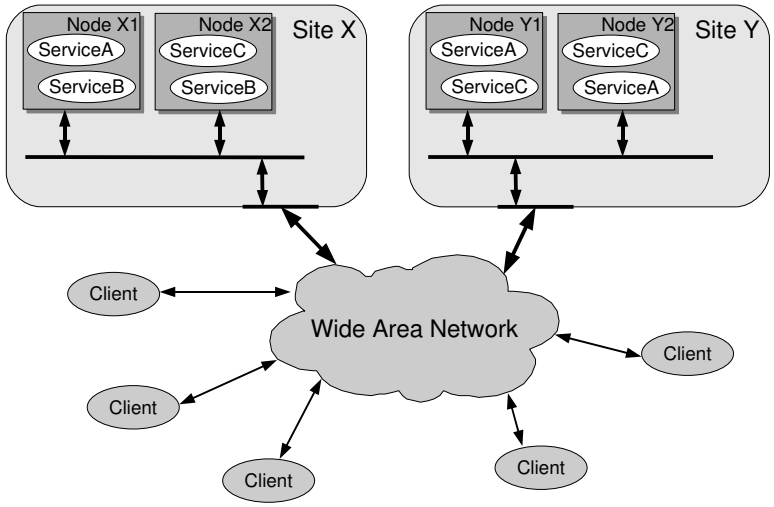


Figure 1.1: An example target environment for a fault tolerant distributed system.

same node. The nodes may be geographically distributed on separate sites to avoid the consequences of a catastrophic failure or a network partition separating clients from all the servers.

Failure assumptions: Objects and nodes in the target environment are assumed to follow the crash (omission) failure semantics, whereas links between nodes/objects may partition and re-merge. The system presented in this thesis is designed to tolerate such failures. Chapter 2 provides a description of these and other failure modes.

1.2.2 Research Methodology

The Jgroup/ARM architecture has been implemented as a proof of concept prototype. Generally, a system or architecture can be evaluated analytically, through simulations, or through measurements on a real system [61]. There are mainly two reasons for doing a prototype implementation:

1. Many middleware platforms already exists, even with support for replication transparency. Hence, a working system is significantly more credible than a system based on simulations or analytical evaluation. However, the drawback of doing a prototype is that it takes longer to implement and test and to perform measurements [61].

2. Both experience and code can be brought into a commercial product based on this work.

The work presented in this thesis follows a traditional research methodology.

Work hypothesis The contributions of this thesis are all based on an initial idea, i.e. a hypothesis. The main ideas in Part III are geared towards techniques aimed at achieving the overall goal of this work, whereas ideas in Part II are needed to support the main goal.

Hypothesis testing To determine the value of the ideas presented in this thesis, various test scenarios have been constructed. The various tests attempt to model realistic failure scenarios that could potentially occur in a real deployment of a system based on Jgroup/ARM. The tests are designed to test a significant portion of the possible ways in which the system can fail, rather than testing only the common failure scenarios. A measurement based evaluation is the natural choice, since a prototype implementation has been developed. Chapter 13 presents the framework used for the tests. This framework has been developed in the context of this thesis, and aims to simplify and automate the execution of experiments.

Result validation Results are obtained from the various test scenarios. Some tests are aimed at revealing the delays involved in fault treatment, whereas others aim to test Jgroup/ARM resilience when exposed to a rapid succession of failures. Few other comparable results exists, but where appropriate a comparison is made.

1.2.3 Contributions

A Revised Object Group Platform for Java The prototype developed in the context of this thesis is based on the Jgroup [87] object group system. The initial Jgroup design followed a rather monolithic approach which made it difficult to support the flexibility and adaptivity required for autonomic management of services. However, other benefits of Jgroup outweighed these considerations, and hence the decision was made to enhance Jgroup with the necessary features to support our requirements. In particular, the revised Jgroup core has been made significantly more flexible by introducing layer-like *modules* that can be configured to interact in various ways. New modules can easily be added, and a number of modules have been implemented to solve specific tasks needed by the other contributions below. Additional flexibility

has also been gained by allowing multiple replicas (of distinct services) to be located on the same node. Furthermore, clients also need to adapt to the dynamics of the environment to provide an adequate failover mechanism in the face of failures and recoveries. Finally, an architecture for configurable replication protocols has been added to simplify application design. These are all issues that needed to be dealt with in order to support our main goal.

The Autonomous Replication Management Architecture We propose the design of an event-driven architecture for *Autonomous Replication Management* (ARM), that provides replication management and recovery to Java-based distributed applications. The ARM architecture is built on top of an object group system and features mechanisms for distributing replicas (geographically) on separate nodes, and handling recovery from replica failures, by means of creating a replacement replica on an alternative node. This is an effective mechanism for rendering replica failures independent.

The Upgrade Management Architecture An architecture for software upgrade management is proposed, ensuring uninterrupted service provisioning during the upgrade process. Upgrade management is incorporated in the ARM architecture and takes advantage of the fact that we can, for a short period of time, decrease the redundancy level of a service to handle the upgrade process by replacing the replicas one by one.

Extensive Experimental Analysis To demonstrate the usefulness of Jgroup/ARM and its impact on service availability, three major experimental evaluations has been conducted. Each experiment focuses on a separate aspect: *client performance and failover latency* in response to server failures, obtaining *service availability metrics* for a service exposed to multiple nearly-coincident crash failures and determining the *recovery performance* for the system when exposed to multiple nearly-coincident partition failures. In the second experiment, a novel technique has been developed to assess the dependability characteristics of a system deployed using Jgroup/ARM.

1.3 Roadmap

The dissertation is organized in five parts as illustrated in Figure 1.2.

Part I gives an overview of the research topics covered in this thesis. Chapter 2 gives a brief state of the art overview of fault tolerant distributed systems and attempts to

relate previous works to the work presented in this thesis. Readers familiar with the field may browse quickly through the chapter or skip directly to Chapter 3.

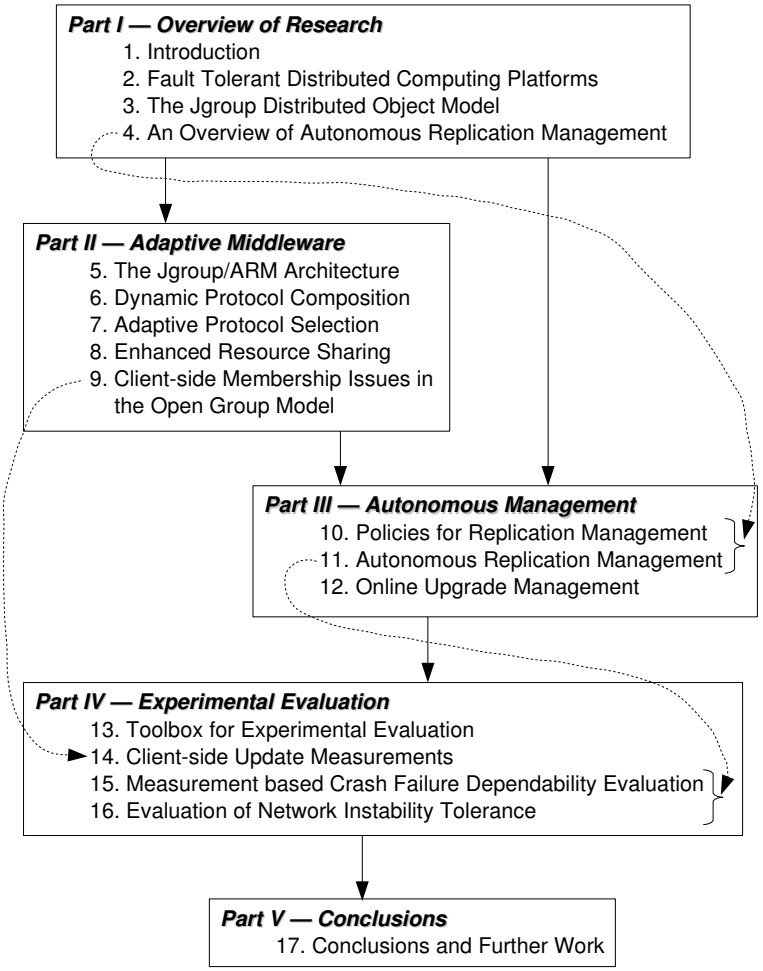


Figure 1.2: Organization of the dissertation.

Chapter 3 gives an overview of the Jgroup distributed object model on which this work is based; the description is included to make the thesis self-contained. Readers familiar with Jgroup may skip to Chapter 4. The main contribution of this work is briefly presented in Chapter 4. It serves to give the reader an overview of the ARM framework before covering the necessary changes to the Jgroup platform to support ARM. The details of the ARM framework is covered in Part III.

Part II covers the numerous enhancements made to the Jgroup platform to support ARM and in general to improve the system. Chapter 5 presents the joint Jgroup/ARM architecture. Chapter 6 describes the new protocol composition framework that is capable of dynamic construction of protocols based on some configuration. The architecture for adaptive protocol selection is covered in Chapter 7. It allows both design time and runtime adaption of replication protocols on a per method basis. Issues concerning enhanced resource sharing among replicas on the same node is covered in Chapter 8. This improvement makes it easier for ARM to place multiple replicas on the same node. Chapter 9 describes techniques to overcome potential membership inconsistencies on the client side of a group communication system based on the open group model, such as Jgroup.

Part III treats the main architectural contributions of this work, the ARM framework. Chapter 10 describes the policy framework in which policies for replication management are defined; the policy framework is used by the various Jgroup/ARM components to determine their self-regulatory behavior. A detailed description of the ARM framework is given in Chapter 11; ARM is a self-managing fault treatment system aimed at improving the dependability characteristics of deployed services. Chapter 12 describes a complementary architecture to enable online upgrading of services by exploiting synergies with ARM.

Three distinct measurements of the ARM framework are covered in Part IV. Initially, in Chapter 13, we describe the experiment framework used to conduct the measurements in the following three chapters. The first set of measurements presented in Chapter 14 aims to reveal the benefit of updating the client-side membership information when invocations are load balanced on a set of servers. The failover latency seen by clients is also measured in this experiment. These measurements are related to the techniques described in Chapter 9. Chapter 15 presents a novel evaluation technique to estimate dependability attributes of a system based on measurements. This technique is applied to a service deployed using Jgroup/ARM, when exposed to crash fault injections. The final experimental evaluation presented in Chapter 16, aims to evaluate the network instability tolerance of the Jgroup/ARM framework. In this experiment, network instability is emulated through injection of one or more network partitions in the system.

Part V concludes the thesis by reviewing the main topics covered herein. Ideas for future work is also outlined.

Chapter 2

Fault Tolerant Distributed Computing Platforms

This chapter gives a brief overview of the *state of the art* in the field of fault tolerant distributed systems and middleware for such systems, and attempts to relate previous works to what has been done in the context of this thesis.

2.1 Distributed Computing Systems

Computer programs that consist of two or more decoupled program components that interact with each other by the exchange of messages, is considered a *networked application.* In contrast, a *distributed computing application* is comprised of a set of tightly coupled program components running on several computers, coordinating their actions [22]. The purpose of building distributed applications is to circumvent the limited resources of a single computer, through exploitation of the aggregate resources of multiple (possibly less powerful) computers. The resources we refer to may be information, disk capacity, CPU cycles and so on. For example, an online banking service, as shown in Figure 2.1, may be implemented as a distributed application involving a large number of clients, and a number of server objects implementing various parts of the banking application. Each of the server objects may be located on distinct nodes within the bank network.

Although distributed computing has many appealing properties, they are very difficult to build and manage correctly. This is due to issues such as synchronization, failures,

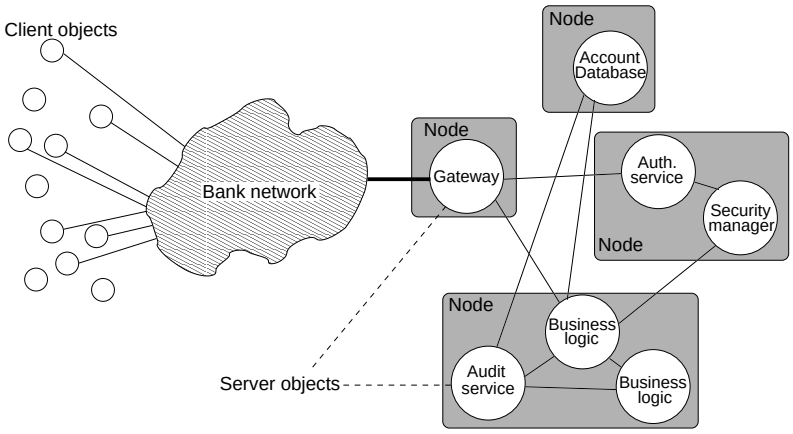


Figure 2.1: An imaginary distributed banking application.

unreliable and insecure communication. The primary aim of middleware platforms for distributed computing is to support the *distribution transparency*, as discussed previously.

2.1.1 Dependable Distributed Systems

Since distributed systems typically involve a large number of hardware and software components, more things can go wrong. That is, without taking additional measures, a distributed system is per se less *dependable* than a centralized system. Dependability is a term that covers many useful requirements for distributed systems [62, 56]:

1. **Reliability** is defined as the system's ability to provide uninterrupted service. In other words, a highly reliable system is one that is likely to continue to work without interruption for a relatively long period of time.
2. **Availability** refers to the property that a system is ready to be used immediately. Thus, a highly available system is one that is likely to be working at a given instance in time.
3. **Safety** refers to a system that may fail temporarily, yet nothing catastrophic happens. For example, a system stop may be considered a benign failure, while loss or corruption of data is considered catastrophic.
4. **Security** concerns the system's ability prevent unauthorized access to data, services and other resources.

In this dissertation, focus is on reliability and availability.

In the dependable system community, *fault*, *error* and *failure* have a specific meanings, whereas in daily language these words are often used interchangeably to mean the same thing. This dissertation use the definitions of [11]:

1. **Failure** Deviation of the delivered service from compliance with the specification. Transition from correct service delivery to incorrect service delivery.
2. **Error** Part of a system state which is liable to lead to failure. Manifestation of a fault in a system.
3. **Fault** Adjudged or hypothesized cause of an error. Error cause which is intended to be avoided or tolerated.

Hence, an error is the manifestation of a fault in the system, whereas a failure is the effect of an error on the service.

There are an infinite number of ways in which a system can fail, and to be able to build distributed systems that tolerate failures, we need to provide a precise and clear definition of the types of failures that the system will tolerate. The various ways in which a system can fail is often referred to as the *failure modes* of the system. Although a system can fail in a number of different ways, it is common to classify the various failure modes as shown in Figure 2.2, and briefly described below.

1. **Value failures** occur when the value of a response from the system implementation does not comply with the system specification. Value failures may either be *consistent*, given the same input to the system, or *inconsistent*. Inconsistent failures are often referred to as **Byzantine failures** [70].
2. **Timing failures** are related to violation of the temporal properties of the system. Timing failures occur when the response to an input arrives too late/early at its destination. The response value may otherwise be correct, but has become invalidated by its late (or early) delivery.
3. **Omission failures** can be viewed as a special case of both value and timing failure, and occur when the system provides no response to the provided input. An omission failure can be either persistent or non-persistent. Persistent omission failure is commonly denoted as **crash failure**, meaning that a unit (e.g. object) simply halts, losing its volatile data.

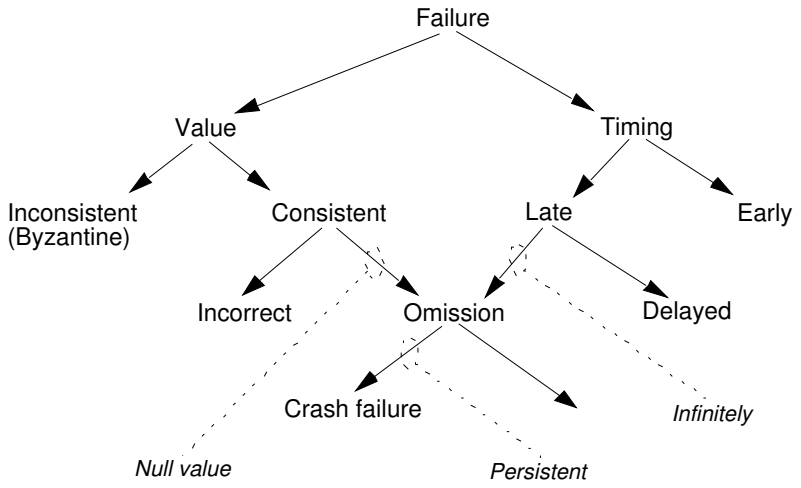


Figure 2.2: Classification of failure modes (adopted from [56]).

In the context of distributed systems, it is common to use a further subdivision of the failure modes that a system can tolerate. In particular, one such failure mode is **network partitioning failure**, which is a special kind of omission failure, and may occur when a network is fragmented into two or more subnetworks that are unable to communicate with each other. A set of communicating processes may perceive a network partition for a number of reasons. Events such as physical link breakage, buffer overflows, incorrect or inconsistent routing tables may disable communication between a pair of objects [14]. Such failures are also referred to as communication failures.

In this work, we consider objects and nodes that follow crash failure semantics, and in addition the network may partition and re-merge.

There are two general approaches to designing dependable distributed systems with emphasis on reliability and availability attributes [56]:

- fault avoidance, and
- fault tolerance.

In this thesis, we focus on techniques related to fault tolerance.

A *fault tolerant* system is one that is able to continue to provide service in spite of the occurrence and activation of faults in the system [56]. Building a fault tolerant system requires some form of redundancy to detect, correct, or mask the effects of faults. In

this work, we consider redundancy in the form of additional system components or replicas of objects.

To further improve the dependability characteristics of fault tolerant systems based on replication, fault treatment can be introduced. A *fault treatment* system is one that is able to reconfigure the system to either rectify or reduce the consequences of a fault/failure. For example, fault treatment can be used to reconfigure the system to restore the redundancy level of a service so that the system is able to tolerate further faults. Fault treatment typically involves three phases: fault diagnosis, fault passivation and system reconfiguration. The fault diagnosis aims to localize the fault and to decide whether fault passivation is necessary to prevent the fault from causing further errors [102]. System reconfiguration often entails the allocation (relocation) and initialization of new replicas to replace failed ones to restore the level of redundancy.

Software components deployed in a distributed system are rarely static, and new versions are often deployed to replace old ones. In most distributed systems such software upgrading requires that the whole system is taken offline while the upgrade takes place. Such a scheme would be severely detrimental to the system's availability. However, when used in conjunction with a fault tolerant system based on replication, *online upgrades* are made possible by replacing replicas one-by-one, while at least some of the replicas (old and new) remains operational and able to service clients. Hence, online upgrades can also be viewed as a means to improve the service availability characteristics, by eliminating (or reducing) the downtime during maintenance activity.

2.2 Object-Oriented Distributed Computing Platforms

Distributed computing platforms, commonly denoted middleware platforms, are software components or libraries that intend to ease the development of distributed computing systems. These software components are *logically* layered below the application and above the operating system, hence the name middleware. They hide many details of distribution through the provision of a high-level programming interface that developers may use. In addition to providing programming interfaces, middleware platforms often include a number of common services that applications can take advantage of to further simplify application development. Examples of common services: naming service, notification service, and transaction management services. Such services can be reused by multiple applications within the same system, and as such becomes important infrastructure components.

In recent years, numerous middleware platforms have evolved [50, 123, 120, 7, 39]. All of these middleware platforms are based on the object-orientation paradigm, and their focus is on the non-functional aspects of a distributed system. Many of these systems provide overlapping services and mechanisms. In the following, the most common distributed computing platforms are discussed. The *remote object model* [125] is prevalent in most e-business middleware architectures, and is also the model used by the Jgroup [87] toolkit on which our prototype implementation is based. Hence, we limit our discussion to the most common middleware platforms based on the remote object model.

2.2.1 CORBA

The Common Object Request Broker Architecture (CORBA) is a specification [50] of an architecture for distributed computing. The specification is drawn up by the Object Management Group (OMG), a non-profit consortium with many industry members. The primary goals of the CORBA specification is to provide a common architecture for developing distributed systems, that will run across *heterogeneous* hardware platforms, operating systems and programming languages.

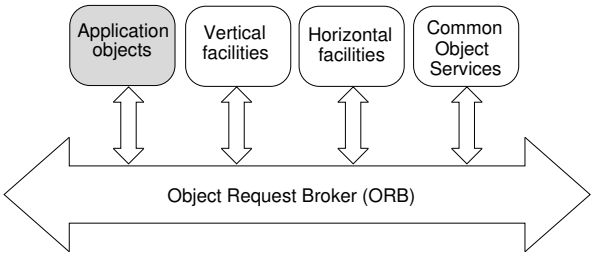


Figure 2.3: The OMG object reference model [50].

The overall architecture of CORBA is laid down in a conceptual model, known as the OMG *reference model*, shown in Figure 2.3. This reference model consists of four groups of architectural facilities that connect to an Object Request Broker (ORB). The ORB is at the core of a CORBA-based distributed system, and enable communication between CORBA objects and clients while concealing issues related to object distribution and heterogeneity. The ORB is also commonly called a communication bus for the CORBA objects, and in many systems the ORB is implemented as a set of libraries that are linked at compile-time with the client and server applications.

The four architectural facilities specified in the reference model is described briefly below:

- **Vertical facilities** are domain specific. That is, they target a specific application domain, such as finance, health care and electronic commerce.
- **Horizontal facilities** consist of high-level general purpose services that are independent of application domain. Examples of such services include document management, printing and task management.
- **Common object services** are basic building blocks commonly used in distributed systems. These include services such as event notification, transaction processing service and naming service.
- **Application objects** are end-user objects that perform specific tasks for a user. A distributed CORBA application may involve a large number of objects, some of which are application objects, while others may be taken from the domain specific, general purpose and common facilities.

Application objects and CORBA services are specified using the CORBA *Interface Definition Language* (IDL). CORBA IDL is a declarative language, derived from C++ syntax, in which methods and their arguments can be specified. However, CORBA IDL has no provision for specifying semantics. It is also necessary to provide rules for mapping an IDL specification to existing programming languages. Currently, such rules exist for a number of languages, including C, C++, Java, Smalltalk, Ada, COBOL, Lisp, Python and IDLscript.

In the common object services, a number of services have already been defined, and new services continuously appear. However, for a long period, CORBA lacked real support for fault-tolerance. That is, a failure of a server would simply be reported to the client, and no further actions were taken by the CORBA system. In CORBA version 3 [50] however, fault-tolerance have been addressed specifically. The specification for fault tolerant CORBA (FT CORBA) can be found in [49]. The basic mechanism for dealing with failures in CORBA is to replicate objects into object groups. The group as a whole can be referenced as if it were a single object, and as such provide transparent replication to its clients. Several replication strategies are supported in the fault tolerant CORBA specification, in particular various incarnations of active and passive replication. In Section 2.5.3 the FT CORBA specification is discussed further from a fault treatment perspective.

At the outset of this thesis, the intention was to enhance CORBA with support for failure transparency and other fault-tolerance mechanisms. However, this intent was

later abandoned for several reasons. Firstly, at the time the only available CORBA based platform supporting fault-tolerance were OGS [41]. Albeit being a very good framework, it had not been maintained for several years and lacked support for recent developments in the CORBA architecture. Furthermore, the ORBs were not designed to cope with deployment of services in a wide-area network, a fundamental requirement if replicas are to be distributed geographically. Finally, the complexities of the CORBA architecture had caused a lot of companies to adopt competitive technologies instead, such as J2EE and EJB in particular.

In discussions with companies providing CORBA implementations it seemed unlikely that they would provide independent implementations of the FT CORBA standard, and would instead rely on the FT CORBA solution provided by Eternal systems to transparently provide fault tolerance for their systems. Eternal systems have since changed their name to Availigent [10] and now provides fault tolerant middleware for ”all“ classes of applications. It seems that their focus has moved away from FT CORBA. Additional details concerning the FT CORBA standard and related technologies are discussed by Felber and Narasimhan in [45].

2.2.2 Java Remote Method Invocations

Java Remote Method Invocations (RMI) is a distributed object model for the Java programming language. It retains as much as possible of the semantics of the Java object model, simplifying the development of distributed objects [136]. A *remote object* in the Java RMI model, is one whose methods can be invoked from another *Java Virtual Machine* (JVM). The invoking JVM may be located on the same local node or a remote node. The methods of a remote object that can be invoked remotely must be declared in a *remote interface*. The invocation of a method on a remote object is referred to as a *remote method invocation*.

Note that the following discussion is adapted from [87] and is slightly more detailed than the other technologies discussed herein; this is motivated by the fact that the Jgroup prototype extends the Java RMI model with support for object groups and group method invocations.

In Java RMI, remote interfaces must satisfy the following requirements:

- A remote interface must at least extend, either directly or indirectly, the interface java.rmi.Remote, which is a marker interface that defines no methods.

- Each method declaration in a remote interface must satisfy the requirements of a remote method declaration as follows:

    1. A remote method declaration must include the java.rmi.RemoteException in its *throws* clause, in addition to any application-specific exceptions. Remote exceptions are thrown when a remote method invocation fails for some reason, such as communication failures (unreachable servers, servers refusing the connection, etc.) or failures during parameter marshaling or unmarshaling.
    2. In a remote method declaration, a remote object declared as a parameter or return value (either declared directly in the parameter list or embedded within a non-remote object) must be declared as the remote interface, not the implementation class of that interface.

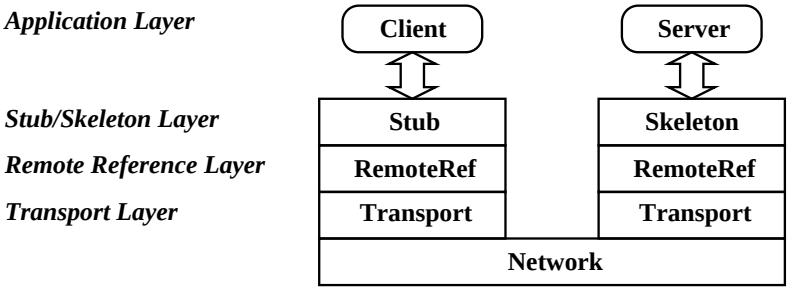


Figure 2.4: The Java RMI Architecture

The Java RMI architecture is illustrated in Figure 2.4. Java RMI uses a standard mechanism (derived from RPC) for communicating with remote objects: *stubs* and *skeletons*. The stub is the client's local representative for the remote object. Clients interact with remote objects through their local representative stub object. The stub is responsible for performing method invocations on their remote counterpart. A skeleton for a remote object is a server-side entity that dispatches invocations on the actual remote object implementation. Stubs and skeletons can be generated by the `rmic` compiler supplied with the standard Java RMI distribution. However, in recent versions of the Java 2 Platform, the stubs and skeletons can also be generated dynamically at runtime, obviating the need to use the `rmic` compiler.

A stub for a remote object implements the same set of remote interfaces implemented by the remote object. Invoking a method on a stub, the stub does the following:

- initiates a connection with the remote JVM containing the remote object;
- marshals (writes and transmits) the parameters to the remote JVM;
- waits for the result of the method invocation;
- unmarshals (reads) the return value or exception returned;
- returns the value to the caller.

In the remote JVM, each remote object have a corresponding skeleton. The skeleton is responsible for dispatching the invocation to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

- unmarshals (reads) the parameters for the remote method;
- invokes the method on the actual remote object implementation;
- marshals the result (return value or exception) to the invoker.

In JDK 1.2 an additional stub protocol was introduced that eliminates the need for skeletons. Instead, generic code is used to carry out the duties of skeletons.

Each stub contains a *remote reference*, that can be seen as a handle for the remote object and is responsible for the semantics of the invocation. The current version of Java RMI includes only two unicast (point-to-point) invocation mechanisms, one relative to servers always running on some machine, and one relative to servers that are activated only when one of their methods is invoked. No multicast invocation mechanism is provided. One of the objectives of Jgroup [87] is to provide a multicast invocation mechanism for remote method invocations. This is discussed at length in Section 3.3.2.2.

Before a client can invoke the methods of a remote object, it must obtain a stub for it. For this reason, the Java RMI architecture includes a repository facility called rmiregistry that can be used to retrieve remote object stubs by simple names. Each registry maintains a set of bindings $\langle name, remote\ object \rangle$; new bindings can be added using method bind, while a lookup method is used to obtain the stub for a remote object registered under a certain name. Since registries are remote objects, the Java RMI architecture includes also a bootstrap mechanism to obtain registry stubs. Jgroup provide a dependable registry, as discussed in Section 3.4.

2.2.3 Jini

The Jini technology framework [121, 122, 7] provides an infrastructure for defining, advertising and finding services in a network. It takes care of some of the common, but difficult, issues of distributed systems development. A Jini system consists of the following parts [7]:

- A set of components that provides an *infrastructure* for federated services in a distributed system.
- A *programming model* that supports and encourages the production of reliable distributed services.
- *Services* that can be made part of a federated Jini system and offer functionality to any other member of the federation.

Figure 2.5 shows the various services, programming models and infrastructure components supported by the Jini architecture.

	Infrastructure	**Programming Model**	**Services**
Base Java	Java VM RMI Java Security	Java API JavaBeans™ ...	JNDI Enterprise beans ...
Java + Jini	Discovery/Join Distributed Security Lookup	Leasing Transactions Events	Printing Transaction Manager JavaSpaces™Service ...

Figure 2.5: Jini Architecture Segmentation [121].

Note that we have used Jgroup to enhance the Jini transaction service with support for replicated transaction managers and participants [85, 66]. Also the Jini lookup service has been enhanced to support object groups [88]. These topics are not covered in this thesis.

2.2.3.1 Jini Extensible Remote Invocation

In recent versions of Jgroup the plain Java RMI model discussed above have been replaced with the more flexible Jini Extensible Remote Invocation (JERI) model [116]. JERI is used to invoke remote methods in Jini. It is designed explicitly for extension of the mechanisms underlying remote invocations. JERI is based on Java RMI, but is more loosely coupled to the Java programming language. This is to improve the interoperability with other languages. There are several implementations of JERI enabling the use of RMI semantics over HTTP, IIOP and also SSL. JERI has also been modified to eliminate the need for compile-time generation of stubs, which is now done using reflection [6] instead. The JERI protocol stack is also more flexible than the Java RMI model, as shown in Figure 2.6.

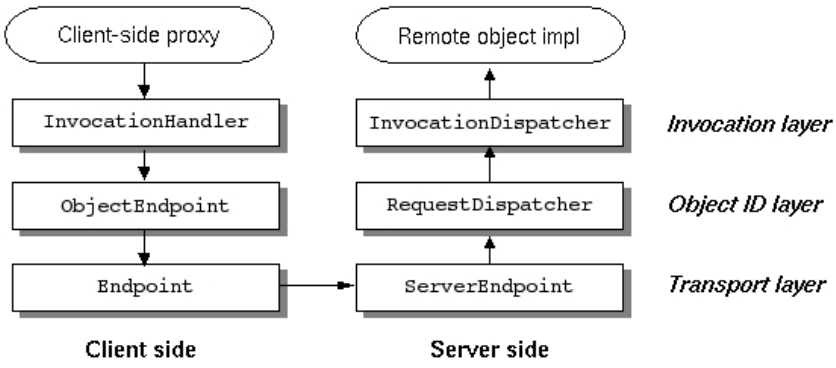


Figure 2.6: The JERI protocol stack [116].

The protocol stack consist of three layers: A transport layer, an object identification layer and an invocation layer. The transport layer is responsible for communicating requests and responses over the network. Remote method invocations are encoded into a transport specific format, which is encapsulated in an Endpoint. There are various implementations of an Endpoint for different types of network transports, for instance TCP, HTTP and SSL. The Endpoint represents the remote communication endpoint to which a request is sent. The object identification layer is used to uniquely identify remote objects. Typically, an object identifier is added to each invocation sent, and the server request dispatcher use this identifier to locate the invocation dispatcher associated with the identifier. The invocation layer is responsible for intercepting calls made by the client, and pass these to the invocation dispatcher

at the server. This includes marshalling and unmarshalling of the objects to be passed back and forth between the client and server. The invocation dispatcher invokes the remote object, and returns any result to the client.

The use of JERI has made it possible to improve the client-side failover mechanism in Jgroup, and also simplified the implementation of the protocol extensions that have been added to Jgroup. These additions are discussed in Part II.

2.2.4 Enterprise Java Beans

The Enterprise Java Beans (EJB) architecture is a multi-tier server-centric component architecture for the development and deployment of component-based distributed business applications [120]. It is one of several coordinated specifications that make up the Java 2 Platform, Enterprise Edition (J2EE). The EJB architecture makes is easy to develop distributed application that are scalable and multi-user secure. It is focused around a transactional model, and provides simple mechanisms for persisting data to persistent storage. The EJB architecture is also portable in the sense that numerous vendors support the EJB architecture in their Application Servers (AS), and hence EJB applications can be deployed in any one of those application servers.

Components developed with EJB technology are often called *enterprise beans*, and typically encapsulate the logic and the data needed to perform operations specific to some business area. A number of distinct enterprise beans are offered, including: *session beans*, *entity beans* and *message-driven beans*.

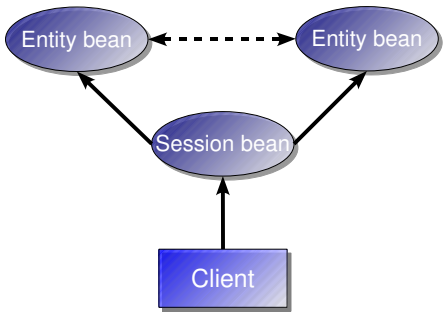


Figure 2.7: The interaction pattern of a simple EJB application.

Clients can interact with the various beans exported by the application. Figure 2.7 illustrates the interaction pattern of a simple EJB application. A session bean instance

is tied to a specific client for the duration of a communication session. Hence, a session bean cannot be shared between multiple clients, whereas the entity bean can be shared between clients. Business tasks to be executed on the server are exposed through methods on the session bean, enabling clients to call these methods.

Communication between clients and stateful session beans usually take place over Java RMI[1]. As mentioned previously, Jgroup extends Java RMI with support for group method invocations. Albeit not tested, this particular feature of Jgroup is expected to enable us to extend the EJB architecture with the fault tolerance mechanisms provided by Jgroup.

2.3 Group Communication Systems

Group communication can be defined as a means for providing multi-point to multi-point communication, through organizing communication entities into groups [29]. Historically, process groups [21, 132, 38] were used, whereas more recent implementations are based on object groups, including Jgroup [87] and OGS [41]. A *group* is defined as a set of communicating objects which are *members* of the group. For example, a group may be a set of server objects (*replicas* for brevity), providing some service to a large number of clients in a highly available and load-balanced distributed system. The object group constitutes a logical addressing facility, allowing external objects, e.g. clients, to communicate with the object group without knowledge of the number of group members, their individual identity and location.

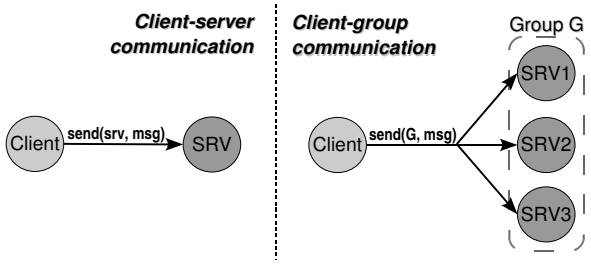


Figure 2.8: Client-server vs. Client-group communication (adapted from [41]).

As depicted in Figure 2.8, messages (or method invocations) are sent from a client

[1]The CORBA IIOP protocol can also be used.

to a group using a single logical address, G, in the same way as in traditional client-server interactions. Hence, to a client the existence of an object group is transparent. Generally, a *group communication system* (GCS) provides two services:

- a *group membership service* (GMS) and
- a *reliable multicast service* (RMS).

A GCS based on an object-oriented approach is often denoted an *object group system* [33, 74, 41, 87]. The interaction primitives provided by the reliable multicast service may support a number of reliability and ordering guarantees. The Jgroup RMS is based on method invocations rather than message passing and is covered in Section 3.3.2. The group membership service is briefly covered in the next section.

2.3.1 The Group Membership Service

An object group is a collection of server objects that cooperate in providing a distributed service. For increased flexibility, the group composition is allowed to vary dynamically as new servers are added and existing ones are removed. A server contributing to a distributed service becomes *member* of the group by *joining* it. Later on, a member may decide to terminate its contribution by *leaving* the group. At any time, the *membership* of a group includes those servers that are operational and have joined but have not yet left the group. System asynchrony and failures may cause each member to have a different perception of the group's current membership. The purpose of the GMS is to track voluntary variations in the membership, as well as involuntary variations due to failures and repairs of servers and communication links. All variations in the membership are reported to members through the *installation* of *views*. A view consist of a membership list along with a unique view identifier, and correspond to the group's current composition as perceived by members included in the view. In response to variations in the group's membership the GMS of all members runs a *view agreement protocol* to reach agreement on a new view.

The GMS provided by Jgroup is a so called partition-aware group membership service (PGMS); a brief informal overview is given in Section 3.3.1.

2.3.2 Primary Partition vs. Partitionable Membership Services

A group membership service may be either *primary partition* or *partitionable*. A primary partition GMS install views at all members in a total order, while in a partitionable GMS the views are only partially ordered [29]. Thus, for a partitionable GMS

multiple concurrent views may coexist in disjoint partitions. A primary partition GMS will, when exposed to a network partition scenario, only allow execution (processing of messages) to continue in a single partition. The other partitions suspend execution until the network becomes connected again. The partition in which execution is allowed is chosen according to some rule, e.g. inclusion of a distinguished node/member or a majority of members [15]. On the other hand, a partitionable GMS allow execution to continue in all partitions. The advantage of this approach is that service is provided to all clients that can reach a server independent of the partition in which they reside. That is, assuming the client does not get partitioned from all the servers.

As discussed in [15], there is a class of applications that can continue to provide service (possibly reduced or degraded) in multiple disjoint partitions. Operation in partitioned mode is dictated by application semantics, e.g. it may be possible to service *read* operations in all partitions, while *write* operations are restricted to the primary partition. An important requirement for partitioned operation is that a common shared state can be reconstructed once the network becomes connected again after a partition scenario. However, for another class of applications, merging the states of disjoint partitions into a single shared state for the common partition is not possible. For example, when two partitions have performed conflicting changes to the shared state. Hence, for applications that require a globally consistent shared state, the primary partition approach should be taken.

The membership service of the Jgroup/ARM system used throughout this work put special emphasis on supporting continued operation in multiple disjoint partitions. However, it is straightforward to enhance the system to notify the members whether they are in a *primary view* or not. This can be used by applications that needs to maintain a globally consistent shared state.

2.3.3 Open vs. Closed Group Communication

Traditionally, group communication systems such as ISIS [21], Horus [132] and Transis [38], have been based on a so called *closed* group model [64]. Thus, when adopting group communication in a client-server setting, the clients are members (or special members) of the server group. The advantage of this approach is that clients have immediate access to the group communication primitives that provide reliable and totally ordered messages. However, this approach is extremely costly when the number of clients increase and it does not scale well [17]. A client could

become member of the group when it needs to interact with the server group, but this approach is also costly with many simultaneous clients. In particular, the view agreement algorithm must be executed for each new client that wish to interact with the group to ensure that all servers deliver the same set of the messages sent by clients. Hence, for client-server systems with a large number of clients that communicate with relatively few servers, the closed group model is unsuitable.

For this reason, many recent group communication systems [87, 41] use the *open* group model [64] instead. In this model, clients need not become members of the server group in order to communicate with it, as illustrated in Figure 2.8. This is the model used by Jgroup [87], and it clearly introduces additional challenges with respect to delivery of messages originated from clients. These issues are discussed further in Section 3.3.2.2.

2.4 Replication Techniques and Protocols

As discussed previously, the use of redundancy to mask the failure of individual components is a common technique to improve the reliability and availability characteristics of a system. In the context of this work, the unit of replication is an object. The set of objects being replicated can be either static or dynamic. Using static replication, the number of replicas and their identity is fixed for the duration of the objects life. Dynamic replication on the other hand, allow replicas to be added or removed at runtime. To support such dynamism of replicas, a group membership service is often used, as discussed above. Various well-known replication techniques and protocols commonly used in systems are presented below.

2.4.1 Active Replication

Active replication – also called *state machine approach* [108] or *modular redundancy* [56] – is a technique in which all replicas process requests from clients and update their states before sending a response to the client (see Figure 2.9). The main advantage of this approach is that since the requests are always sent to all replicas, it is possible to mask replica failures from the client as long as there are at least one live replica capable of responding to the client. Hence, client requests are processed uninterrupted; there are no delays incurred by any recovery mechanism.

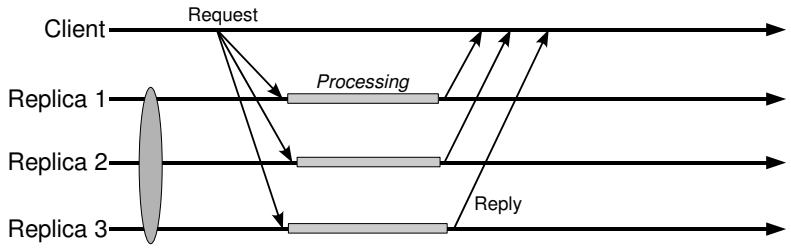


Figure 2.9: Active replication.

The main drawback with this approach is that all replicas must process every request and return a reply in a deterministic manner. There are many sources of non-determinism in a program [101]. For instance, building a multi-threaded server that is actively replicated is a challenging task [98], since multi-threading is inherently non-deterministic in its scheduling of threads. Also, since all replicas have to process every request, and respond, the technique naturally has a high resource consumption, both with respect to CPU and communication. In addition, to maintain strong consistency, communication intensive protocols are required, e.g. atomic multicast (see Section 2.4.6).

There are two kinds to active replication, depending on the severity of the failures to be tolerated as discussed next.

Active replication without majority vote If we can assume failstop [107] behavior of replicas only $f + 1$ replicas are needed to tolerate f failures.

Active replication with majority vote Given that Byzantine failures are allowed, an f fault-tolerant system must have at least $3f + 1$ replicas. A majority vote has to be performed on the result from all the $3f + 1$ replicas, allowing correct output even with f failures. With $f + 1$ failures there are not enough correct replicas to make a majority decision, since Byzantine failures are possible.

Note that active replication requires that client requests are sent *directly* to all replicas, and that each replica respond *directly* to the client. Hence, it would be beneficial to exploit multicast directly from clients to achieve this [118]. However, since clients tend to be located at sites different from those of the servers, this may not be possible due to the limited deployment of IP multicast in Internet routers. For this reason, Jgroup takes a different approach when a client needs to communicate with all replicas as discussed in Section 3.3.2.2.

2.4.2 Passive Replication

Passive replication – also called the *primary-backup approach* [26, 51] or *standby redundancy* [56] – is a technique in which only one replica (the primary) processes requests from clients, and send their state to the other replicas (the backups) (see Figure 2.10). Note that the primary does not return a reply to the client until the state of the backup replicas have been brought up-to-date. Hence, passive replication provides strong consistency between the replicas. In response to the failure of the primary, a backup will be selected to take over the responsibility of the primary. If the primary fails during the processing of a request, it is the client's responsibility to reissue the request to a new primary.

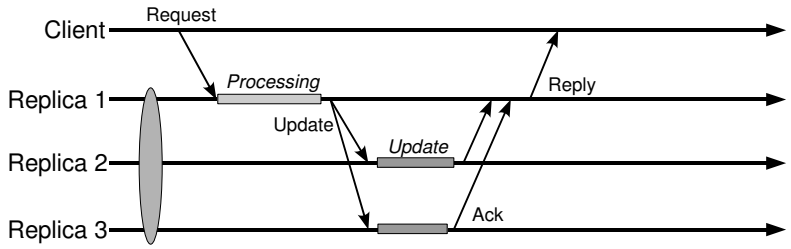


Figure 2.10: Passive replication.

The benefits of passive replication is that less processing power is needed, while still being able to provide fault tolerance. But perhaps the most important advantage of passive replication is that processing can be non-deterministic. The main drawback with this approach is that it has slow reaction to failures, and hence may be unsuitable for certain time-critical systems. This is since most implementations of passive replication rely on a GMS to exclude the primary from the membership when it is suspected, and often the view agreement protocol is configured with a conservative timeout value, to avoid false suspicions [36].

A variant of the passive replication protocol is implemented in Jgroup, and is discussed in Section 7.4.

2.4.3 Semi-Active Replication

Semi-active replication was developed in the context of Delta-4 [102, 20], which was designed specifically to support time-critical systems and assumes a synchronous

system model. The technique extends active replication with the notation of leader and followers. The actual processing is performed by all the replicas, however, only the leader performs the non-deterministic parts of the processing and inform the followers. Hence, semi-active replication circumvents the requirement for deterministic processing in active replication. With respect to computational resources the technique is equivalent to active replication, since all replicas process the requests.

2.4.4 Semi-Passive Replication

Semi-passive replication [37, 36] has the same advantages as the passive replication technique, but is claimed to have faster reaction time to failures than passive replication. This is accomplished by using a separate consensus protocol to elect a new primary with a more aggressive timeout value than what a group membership service typically allow.

2.4.5 Combining Replication Techniques

Combining several replication techniques or protocols in a common framework is appealing since it contributes significant flexibility to the developer and may also reduce the overall cost of replication. Such hybrid replication techniques are known from hardware fault tolerant systems [111].

The approach proposed in [76] suggests to combine active and passive replication. Assuming a redundancy level of R, then only $R_{active} \leq R$ replicas are active, while the remaining $R - R_{active}$ replicas are passive. In response to failure of an active replica, one of the passive replicas can be promoted to become an active replica. The objective of the technique is to reduce the number (R_{active}) of active replicas that needs communication intensive protocols to maintain strong consistency, hence obtaining a performance gain without reducing the service availability. This technique has not been pursued by a prototype implementation due to its complexity and lack of required support in the middleware platforms that existed at the time.

In [42] an approach to combining active and passive replication is presented, allowing the server side to dynamically assign distinct replication protocols to each individual operation/method of the replicated object. What this means is that individual methods may use different protocols, some of which are weaker but more efficient, while others are stronger but less efficient. A similar approach is used in our protocol framework discussed in Chapter 7.

2.4.6 Atomic Multicast

Atomic multicast [54] – also called *total order multicast* – is perhaps the most important protocol for building replicated services, since it facilitates maintenance of a globally consistent shared state. Put simply, atomic multicast requires that all correct objects deliver *all* messages in the same order. This total order message delivery ensure that all objects have the same view of the system, facilitating consistent behavior without additional communication [54]. Figure 2.11 illustrates two clients, each sending a message to a group of replicas, with and without total ordering. As shown, when total order is enforced, the delivery of some messages (m_2 in this case) may have to be delayed.

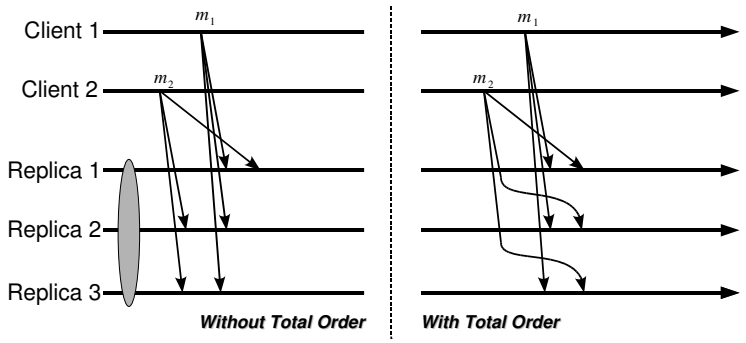


Figure 2.11: Total order multicast.

A variant of the ISIS total ordering protocol [23] has been implemented in the context of this thesis (see Section 7.5).

2.5 Dependable Middleware Platforms

2.5.1 Classification of Dependable Middleware

Felber [41] introduced a classification of CORBA based group communication systems into three broad categories: integration, interception and service. This classification can aid us in identifying the intrusiveness imposed on the developer of distributed applications.

1. **Integration Approach** The *integration approach* entails modifying and enhancing the ORB using an underlying group communication system. CORBA invocations are passed to the group communication service that multicasts them to replicated servers. This approach was pursued by the Electra system [73].

2. **Interception Approach** In the *interception approach*, low-level messages containing CORBA invocations and responses are intercepted on client and server sides and mapped onto messages sent through a group communication system. This approach does not require any modification of the ORB, but relies on OS-specific mechanisms for request interception. Eternal [93, 97] is based on the interception approach.

3. **Service Approach** Felbers last class is the *service approach*. It provides group communication as a separate CORBA service; that is the group communication primitives are embedded into an application programming interface (API) that ease the development of dependable distributed applications. The ORB is unaware of object groups, and the service can be used in any CORBA-compliant implementation. The service approach has been adopted by object group systems such as OGS [41, 43], DOORS [30], TRANSORG [124] and Newtop [91]. Also the FT CORBA standard [49] is based on the service approach.

The main advantages of the first two approaches are that they provide a high degree of *transparency* to the application developer and they have better *performance*, but in general these are not as *portable* as the service approach [41].

Albeit these coarse classes are focused on CORBA related systems, they can also be applied to other middleware architectures. For instance, Aroma [96] is a fault tolerance framework for Java RMI which is based on the interception approach, whereas JavaGroups [18], Filterfresh [19] and Jgroup/ARM [81] are all based on the service approach. However, Jgroup/ARM does exploit several advanced language-level features of Java, such as reflection and annotation [6], to make application development as seamless as possible from a non-functional point of view, by hiding most of the fault tolerance mechanisms. For example, Jgroup provides complete client transparency, while server implementations often needs to provide state transfer/merge operations. In addition, server implementations can simply annotate each method with the replication protocol to use. This particular feature is discussed in Chapter 7.

2.5.2 Object Group Systems

In the following, the most well-known object group systems that can support Java applications are briefly discussed.

The Object Group System (OGS) [41, 43] implements object groups on top of off-the-shelf CORBA ORBs through a number of services. The services provided by OGS includes reliable unordered multicast, monitoring, group membership and a consensus service. The consensus service is used to implement total ordering. OGS supports group method invocations from clients, much like the approach taken by Jgroup. On the other hand, servers must interact through message multicasting instead of method invocations; Jgroup servers can interact through method invocations.

The TRANSORG system [124] implements the FT CORBA standard by extending an existing ORB using portable interceptors, but also provides an alternative implementation called FA-CORBA which avoids that certain infrastructure components becomes single points of failure.

Filterfresh [19], like Jgroup, integrates the object group paradigm with the Jgroup distributed object model. Filterfresh is rather limited, as it does not provide support for multicast semantics and many other features needed by modern distributed applications.

JavaGroups[2] [18] is a message-based group communication system written in Java providing reliable multicast communication. JavaGroups can support remote method invocations, but not transparently. This is since it is based on the exchange of objects that encode method invocation descriptions. The JBoss Application Server includes support for Enterprise Java Beans, and also supports clustering based on the JavaGroups toolkit [68].

Also Spread [3] is a message-based group communication system implemented in C, but it provides a Java API. Like Jgroup, Spread is also designed especially for partition-awareness and wide-area networks. Some recent dependability middleware architectures have been built on Spread, including MEAD [105], discussed in the next section.

Aroma [96] provides transparent fault tolerance to Java RMI through a partially OS-specific interception approach similar to that of Eternal [93], and relies on an underlying group communication system (Totem [92]) implemented on the native operating

[2]Note that JavaGroups were renamed to JGroups (with capital G and ending with an s) around August 2003. Note that this framework should not be confused with the Jgroup toolkit used in this thesis. For this reason, we continue to use the original JavaGroups name when referring to it.

system. Aroma provides a failover mechanism for clients, similar to the approach used in Jgroup. Aroma does not support partition-awareness. Furthermore, the reliable registry implementation included with Aroma is different from the Jgroup dependable registry (see Section 3.4) in that it requires all nodes to run a local registry instance. This approach is not scalable, since the number of registries that needs to be kept synchronized grows with the number of servers deployed in the distributed system.

Jgroup [87] is based on RMI communication and is implemented in its entirety in Java, and does not rely on any underlying toolkits or a special operating system. A particularly attractive feature of Jgroup is its support for transparent client-server group interactions, including transparent failover. Being based on RMI rather than message multicasting also simplifies the adoption of Jgroup as the fault tolerance framework for middleware environments based on Java RMI, such as Jini [7] and EJB [120]. For instance, Jgroup has been used to enhance the Jini transaction manager with replication support [85, 66]. Another distinguishing feature of Jgroup is its focus on supporting highly-available applications to be deployed in partitionable environments. Most of the existing object group systems [43, 30, 19, 96] are based on the primary-partition approach and thus cannot be used to develop applications capable of continuing to provide services in multiple partitions. Very few object group systems [91, 93, 3] abandon the primary-partition model but do not provide adequate support for partition-aware application development.

2.5.3 Fault Treatment Systems

Many middleware platforms support replication, most of which are based on object groups as discussed above. To compliment such middleware platforms and to improve the dependability characteristics of services, fault treatment can be introduced.

Fault treatment techniques were first introduced as part of the Delta-4 project [102]. Delta-4 was developed in the context of a fail-silent network adapter, ensuring that crashed nodes remain silent towards the network. Thus, a fault that results in a node crash is automatically passivated by the very notion of fail-silence. Faults that result in violation of the fail-silence assumption can only be detected if active replication is employed. For such faults, the violating node is assumed faulty and removed from the system as if it had crashed.

None of the Java-based fault tolerance frameworks discussed in the previous section supports fault treatment mechanisms. The FT CORBA standard [49] does specify

certain mechanisms such as a generic factory, a replication manager and a fault monitoring architecture, that can be used to implement a fault treatment facility. However, fault treatment is not covered explicitly in the standard.

Eternal [93, 97] is probably the most complete implementation of the FT CORBA standard. It supports allocating replicas to nodes, however, the exact workings of their approach has (to the author's knowledge) not been published.

DOORS [30, 99] is a framework that provides a partial FT CORBA implementation, focusing on passive replication. It uses a centralized ReplicaManager to handle replica placement and migration in response to failures. The ReplicaManager component is not replicated, and instead performs periodic checkpointing of its state tables, limiting its usefulness since it cannot handle fault treatment of other applications when the ReplicaManager is unavailable.

Also the MEAD [105] framework implements parts of the FT CORBA standard, and supports recovery from node and process failures. However, recovery from a node failure require manual intervention to either reboot or replace the node, since there is no support for relocating the replicas to other nodes. Another part of MEAD [100] is designed to proactively install replacement replicas on the same node in response to fault indications such as process memory exhaustion.

The AQuA [104, 103] framework is also based on CORBA and was developed independently of the FT CORBA standard. Like Delta-4, AQuA also supports passivation of replicas due to value faults.

The approach in [25] focus on fault treatment to improve the dependability of COTS and legacy-based applications. The supported fault types include hardware-induced software errors, process/node crashes and hangs, and errors in the persistent stable storage. Several different levels of fault treatment can be applied depending on the number and severity of faults.

The ARM framework presented herein is a fault treatment system that can tolerate *object*, *node* and *partition* failures. What distinguishes the ARM approach from the above systems is its support for recovery in partition failure scenarios, and its use of policies to facilitate a self-managed system. For example, the main policy used in ARM tries to maintain a specified minimal redundancy level in each network partition that may arise.

2.6 Summary

Distributed computing systems are decoupled program components, that interact with each other by the exchange of messages to perform a task. The major focus in the field of distributed computing in the last decade has been the development of middleware platforms for such systems, and more recently security and reliability aspects have also gained a lot of attention.

As documented in this chapter, there are still open issues and several improvements possible with respect to fault treatment and replication and upgrade management issues. In particular, further reducing the complexity, development efforts and human interactions needed by systems. This thesis seeks to take a step in this direction.

Chapter 3

The Jgroup Distributed Object Model

Building distributed applications that deal with partial component failures is an error-prone and time-consuming task, and developing such applications become even more complex when having to deal with network partition failures. This is because the application has to consider the fact that server replicas residing in distinct partitions may potentially evolve their states in an inconsistent manner. Application complexity is further complicated, by the fact that the application has to deal with replica recovery and deployment issues, such as where to place the server replicas to ensure application dependability. The latter issue is discussed in Chapter 4.

The aim of Jgroup [87] is to provide systematic support for the development of dependable distributed applications in a partitionable environment, by providing an object-oriented development framework.

To make our description self-contained, the following repeats briefly the Jgroup distributed object model and the main services provided by Jgroup [80, 81, 87, 89]. The description in this chapter is more or less the status of Jgroup at the time it was adopted as the base platform for our studies of fault treatment mechanisms. We begin with a short description of our underlying system model in Section 3.1, and in Section 3.2 we give an architectural overview and present the various Jgroup components. Section 3.3 discusses some of the details of the Jgroup services. Finally, in Section 3.4 we discuss the dependable registry provided with Jgroup.

3.1 System Model

The context of this work is a distributed system composed of client and server objects interconnected through a network. The client and server objects are hosted within *Java Virtual Machines* (JVMs). Each JVM can host a bounded (by memory) number of objects, and an application is typically composed of large number of such objects. A *server object* is an object whose methods may be accessed remotely, through the use of *remote method invocations*. A *client object* is one that perform remote method invocations on the server object. Even though we distinguish between client and server objects, there is nothing restricting a server object in assuming the role of client towards another server. As such we may get a chain of client-server invocations, and this is typically called a *multi-tiered architecture*. Figure 3.1 illustrates a typical three-tiered architecture.

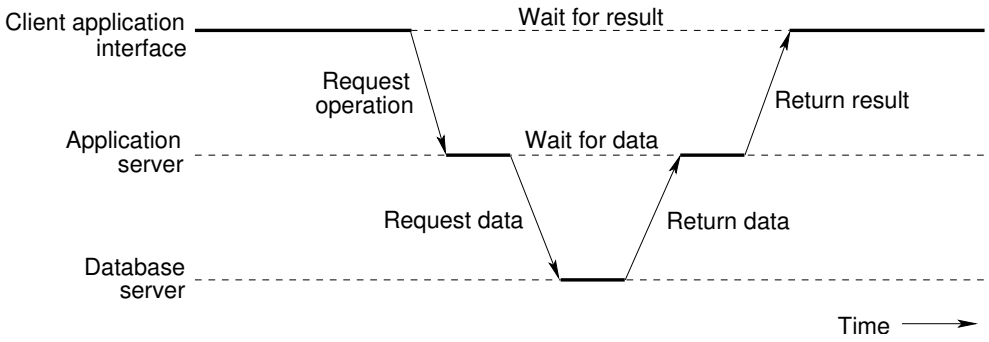


Figure 3.1: Illustration of a three-tiered architecture involving a client application, an application server (business logic) and a database server. The figure is adapted from [125, Ch. 1.5.3].

The distributed system is considered to be *asynchronous* in the sense that neither the computational speed of objects nor communication delays are assumed to be bounded. Furthermore, the system is unreliable and failures may cause objects and communication channels to *crash* whereby they simply stop functioning. Once failures are repaired, they may return to being *operational* after an appropriate recovery action. Finally, the system is *partitionable* in that certain communication failure scenarios may disrupt communication between multiple sets of objects forming *partitions*. Objects within a given partition can communicate among themselves, but cannot communicate with objects outside the partition. When communication between partitions is re-established, we say that they *merge*.

Developing dependable applications to be deployed in these systems is a complex and error-prone task due to the uncertainty resulting from asynchrony and failures. The desire to render services partition-aware to increase their availability adds significantly to this difficulty. Jgroup/ARM have been designed to simplify the development of partition-aware, dependable applications by abstracting complex system events such as failures, recoveries, partitions, merges and asynchrony into simpler, high-level abstractions with well-defined semantics.

Jgroup enable dependable application development through replication, based on the *object group* paradigm [33, 74]. In this paradigm, distributed applications are replicated among a collection of server objects that form a group in order to coordinate their activities and appear to clients as a single server. See Figure 3.2 for a simple illustration.

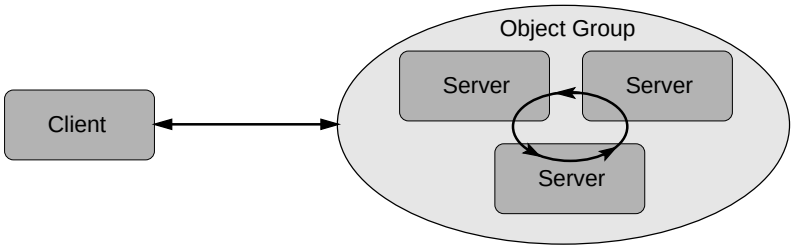


Figure 3.2: Client-to-object group interaction and group internal interactions.

3.2 Architectural Overview

Jgroup extends the object group paradigm to partitionable systems through three core services aimed at simplifying the coordination among replicas: a *partition-aware group membership service* (PGMS), a *group method invocation service* (GMIS) and a *state merging service* (SMS) [87]. These services are briefly described in Section 3.3. An important aspect of Jgroup is the fact that properties guaranteed by each of its components have formal specifications, admitting formal reasoning about the correctness of applications based on Jgroup [87].

As discussed in Section 2.3.1, the task of the PGMS is to provide servers with a consistent view of the group's current membership, to be used to coordinate their actions.

Reliable communication between clients and the object group take the form of *group method invocations* (GMI) [87], that result in methods being executed by the servers forming the group. To clients, GMI interactions are indistinguishable from standard remote method invocations (RMI): clients interact with the object group through a *client-side group proxy* that acts as a representative object for the group, hiding its composition. The group proxy maintains information about the servers composing the group, and handle invocations on behalf of clients by establishing communication with one or more servers and returning the result to the invoker. On the server side, the GMIS enforce reliable communication among replicas.

Finally, the task of SMS is to support developers in re-establishing a global shared state when two or more partitions merge. Servers are called back in order to obtain information about their current state and diffuse them to other partitions.

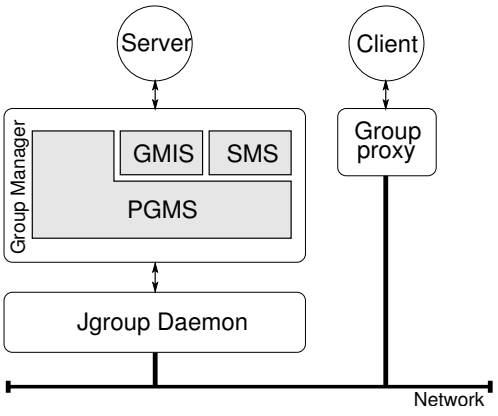


Figure 3.3: Overview of the Jgroup service architecture.

Figure 3.3 gives a high-level overview of the composition of the core Jgroup services. The main component of Jgroup is the *Jgroup daemon* (JD); it implements basic group communication services such as failure detection, group membership and reliable communication. Server replicas must connect to a Jgroup daemon to gain access to the group communication services. Each server replica is associated with a *group manager* (GM), whose task is to act as an interface between the Jgroup daemon and the replica.

Finally, in order to enable clients to locate server groups, Jgroup include the *dependable registry* (DR), a replicated naming service that allows dynamic groups of replicas to register themselves under the same name using the bind() method. Information about the replicas composing the group are collected in a group proxy (GP),

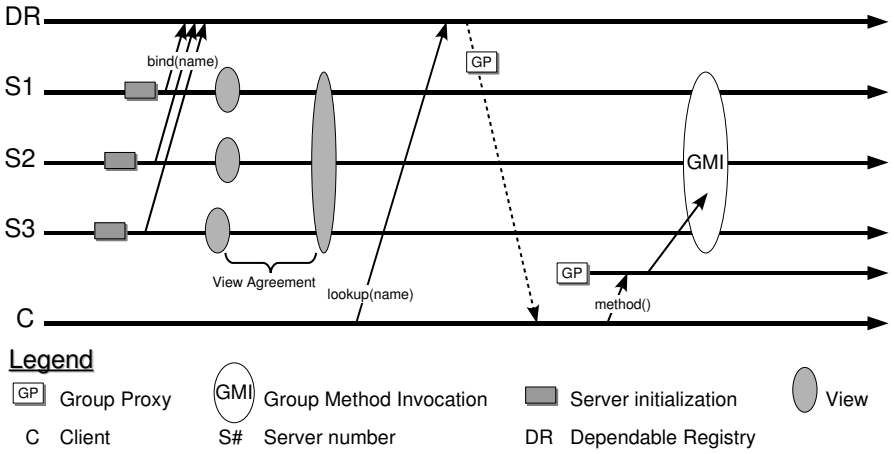


Figure 3.4: Dependable registry interactions with clients and servers.

that can be retrieved by clients using the lookup() method. This enables clients to seamlessly communicate with the whole group as a single entity, by performing invocations through the group proxy. Figure 3.4 illustrates these interactions. Note that the dependable registry itself is implemented as a distributed service, replicated using Jgroup services, although the figure depicts it as a single entity.

3.3 Jgroup Services

The services described below were originally implemented as a (partially) monolithic group manager, while now these have been redesigned using a configurable protocol composition framework as discussed in Chapter 6.

3.3.1 The Partition-aware Group Membership Service

Recall the general description of the group membership service given in Section 2.3.1. A useful PGMS specification has to take into account several issues (see [14] for a detailed discussion of the problem). First, the service must track changes in the group membership accurately and in a timely manner such that installed views indeed convey recent information about the group's composition within each partition. Next, it is required that a view be installed only after agreement is reached on its composition among the servers included in the view. Finally, PGMS must guarantee that two

views installed by two different servers be installed in the same order. These last two properties are necessary for servers to be able to reason globally about the replicated state based solely on local information, thus simplifying significantly their implementation. Note that the PGMS defined for Jgroup admits coexistence of concurrent views, each corresponding to a different partition of the communication network, thus making it suitable for partition-aware applications. Figure 3.5 illustrates the behavior of the PGMS in response to various failure scenarios.

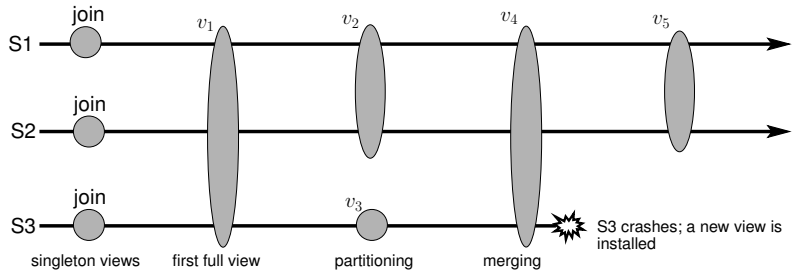


Figure 3.5: PGMS behavior. Servers $S1$, $S2$ and $S3$ join the group, forming view v_1; immediately after, servers $S1$ and $S2$ are partitioned from $S3$. The PGMS reacts by installing two views v_2 and v_3. Later, the partitioning disappears, and nodes are enabled to form again a view v_4 including all members. Finally, server $S3$ crashes, causing view v_5 to be installed including only the surviving members.

3.3.2 The Group Method Invocation Service

Jgroup differs from existing object group systems due to its uniform communication interface based entirely on GMI. Clients and servers interact with groups by remotely invoking methods on them. In this manner, benefits of object-orientation such as abstraction, encapsulation and inheritance are extended to internal communication among servers. Although they share the same intercommunication paradigm, we distinguish between *internal GMI* (IGMI) performed by servers and *external GMI* (EGMI) performed by clients. There are several reasons for this distinction:

- *Visibility:* Methods to be used for implementing a replicated service should not be visible to clients. Clients should be able to access only the "public" interface defining the service, while methods invoked by servers should be considered "private" to the implementation.

- *Transparency:* Jgroup strives to provide an invocation mechanism for clients that is completely transparent with respect to standard Java RMI. Hence, clients should not be aware that they are invoking a method on a group of servers, as opposed to a single server. On the other hand, servers have different requirements for group invocations, such as obtaining a result from each server in the current view.

- *Efficiency:* Having identical specifications for external and internal GMI would have required that clients become members of the group, resulting in poor system scalability. Therefore, Jgroup follow the open group model [64], allowing external GMI to have slightly weaker semantics than those of internal GMI. Recognition of this difference results in a much more scalable system by limiting the higher costs of full group membership to servers, which are typically far fewer in number than clients [17].

When developing dependable distributed services, internal methods are collected to form the *internal remote interface* of the server object, while external methods are collected to form its *external remote interface*. The group proxy objects that are able to handle GMI are generated dynamically (at runtime) based on the remote interfaces of the server object. A proxy object implement the same interface as the group for which they act as a proxy, and enable clients and servers to communicate with the entire group of servers using local invocations on the proxy object. Figure 3.6 details the use of proxies on both the client and server side, as part of the inner workings of the external group method invocation with multicast semantic. That is, a client invoking a multicast method exported externally by the server group.

In order to perform an internal GMI, servers must obtain an appropriate group proxy from the Jgroup runtime running in the local JVM. Clients that need to interact with a group, on the other hand, must request a group proxy from a *registry service*, by performing a lookup operation for the desired service (identified by name) on the registry. The registry enables multiple servers to register themselves under the same service name, so as to compose a group of servers providing the same service. Jgroup feature two different registry services. The first, called *dependable registry* [86], is derived from the standard registry included in Java RMI, while the second is based on the Jini lookup service [7]. The dependable registry service is an integral part of Jgroup and is replicated using Jgroup itself. In this work, only the dependable registry service is used (see Section 3.4).

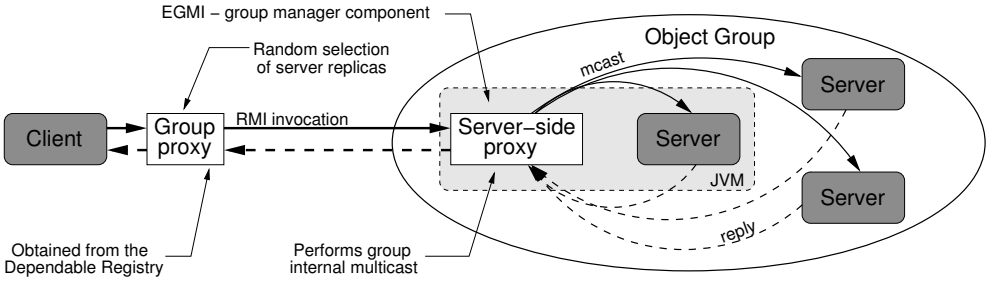


Figure 3.6: Details of the proxy usage in external group method invocation with multicast semantic. The client performs its invocation on the group proxy (obtained from the dependable registry.) The proxy will then perform a plain RMI invocation on one of the servers (the choice of server is random.) Since the invocation has multicast semantic, the receiving server-side proxy will forward the invocation to all servers, and return the result to the client-side proxy. Each server group member must bind with the dependable registry, which can generate a (client-side) group proxy based on its set of known members.

In the following sections, we discuss how internal and external GMI in Jgroup work, and how internal invocations substitute message multicasting as the basic communication paradigm. In particular, we describe the reliability guarantees provided by the two GMI implementations. They are derived from similar properties previously defined for message delivery in message-based group communication systems [14]. In this context, we say that an object (client or server) *performs* a method invocation at the time it invokes a method on a group; we say that a server object *completes* an invocation when it terminates executing the associated method.

3.3.2.1 Internal Group Method Invocations

Unlike traditional Java remote method invocations, IGMI returns an array of results rather than a single value. IGMI comes in two flavors: *synchronous* and *asynchronous*, as illustrated in Figure 3.7.

For synchronous IGMI, the invoker remains blocked until an array containing the result from each server that completed the invocation can be assembled and returned to the invoker. There are many situations in which such blocking may be too costly, as it can unblock only when the last server to complete the invocation has returned its result. Furthermore, it requires programmers to consider issues such as deadlocks

that may occur due to circular invocations. For these reasons, in asynchronous IGMI the invoker does not block, but instead specifies a *callback* object that will be notified when return values are ready from servers completing the invocation.

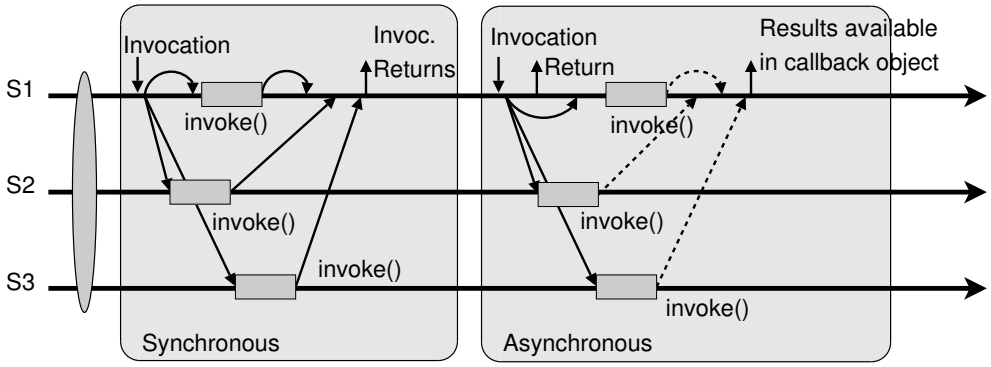
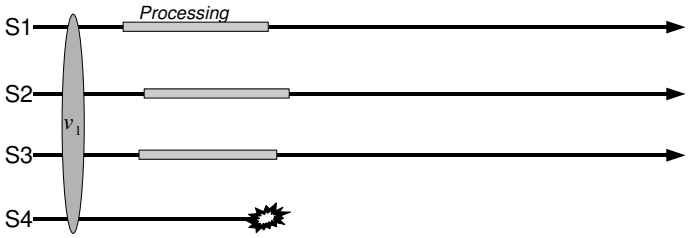


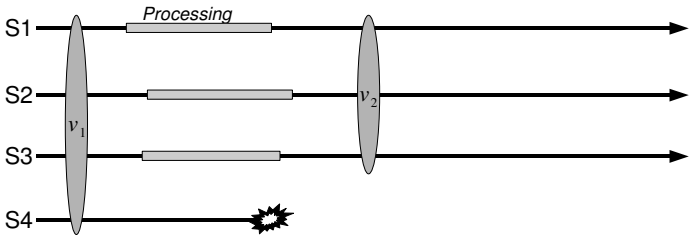
Figure 3.7: In synchronous IGMI, the invoking server is delayed until all servers have returned a result (or an exception). In asynchronous IGMI the invocation returns immediately. However, result values are not readily available, but may later be obtained through the callback object.

If the return type of the method being invoked is void, no return value is provided by the invocation. The invoker has two possibilities: it can specify a callback object to receive notifications about the completion of the invocation, or it can specify null, meaning that it is not interested in knowing when the method completes.

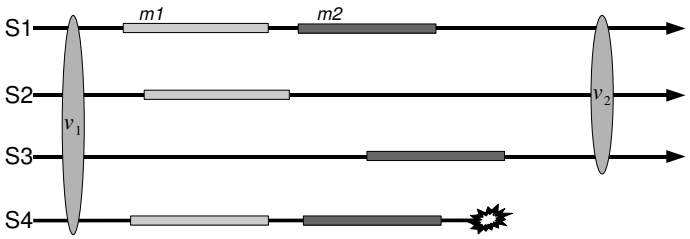
Completion of IGMIs by the servers forming a group satisfies a variant of *view synchrony* that has proven to be an important property for reasoning about reliability in message-based systems [21]. Informally, view synchrony requires two servers that install the same pair of consecutive views to complete the same set of IGMIs during the first view of the pair. In other words, before a new view can be installed, all servers belonging to both the current and the new view have to agree on the set of IGMIs they have completed in the current view. Figure 3.8(a) illustrates a run that violates the view synchrony property, since it does not install a new view after a member has crashed, and the remaining members has processed an IGMI. Figure 3.8(b) illustrates the same run that satisfies view synchrony. Figure 3.8(c) shows another run that violate the view synchrony property. In this case, the various members deliver different sets of IGMIs before installing a new view, clearly violating view synchrony. Figure 3.8(d) illustrates the same run, satisfying view synchrony. Ensuring the view synchrony property enables a server to reason about the state of other servers in the



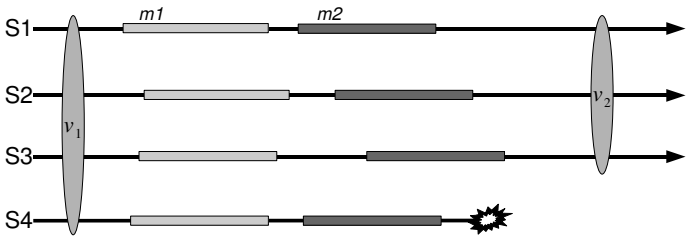
(a) Invalid view synchrony execution.



(b) A valid view synchrony execution.



(c) Another invalid view synchrony execution.



(d) A valid view synchrony execution.

Figure 3.8: Some examples of valid and invalid view synchrony executions.

group using only local information such as the history of installed views and the set of completed IGMIs. Note that view synchrony does not require that the set of IGMIs be ordered. Clearly, application semantics may require that servers need to agree not only on the set of completed IGMIs, but also the order in which they were completed. The standard Jgroup multicast service can easily be enhanced with other multicast services, supporting different ordering semantics. Thus, various ordering semantics can be ensured also for IGMI.

We now outline some of the main properties that IGMI satisfy. First, they are *live*: an IGMI is guaranteed to terminate either with a reply array (containing at least the return value computed by the invoker itself), or with an application-defined exception declared in the *throws* clause of the method being invoked. Furthermore, if an operational server S completes some IGMI in a view, all servers included in that view will also complete the same invocation, or S will install a new view. Since installed views represent the current failure scenario as perceived by servers, this property guarantees that an IGMI will be completed by every other server that is in the same partition as the invoker. IGMI also satisfy *integrity* requirements whereby each IGMI is completed by each server at most once, and only if some server has previously performed it. Finally, Jgroup guarantees that each IGMI be completed in at most one view. In other words, if different servers complete the same IGMI, they cannot complete it in different views. In this manner, all result values that are contained in the reply array are guaranteed to have been computed during the same view.

3.3.2.2 External Group Method Invocations

The EGMI approach supports two distinct invocation semantics, namely: *anycast* and *multicast*, as illustrated in Figure 3.9. The anycast EGMI is performed by a client on a group and will be completed by at least one server of the group, unless there are no operational servers in the client's partition. Anycast invocations are suitable for implementing methods that do not modify the replicated server state, as in query requests to interrogate a database. Multicast EGMI performed by a client on a group will be completed by every server of the group that is in the same partition as the client. Multicast invocations are suitable for implementing methods that may update the replicated server state.

Notice how the multicast invocation semantic differ from an active replication protocol (see Section 2.4.1). In active replication, the client communicates directly with all the servers, whereas in our multicast approach, the client only communicates with

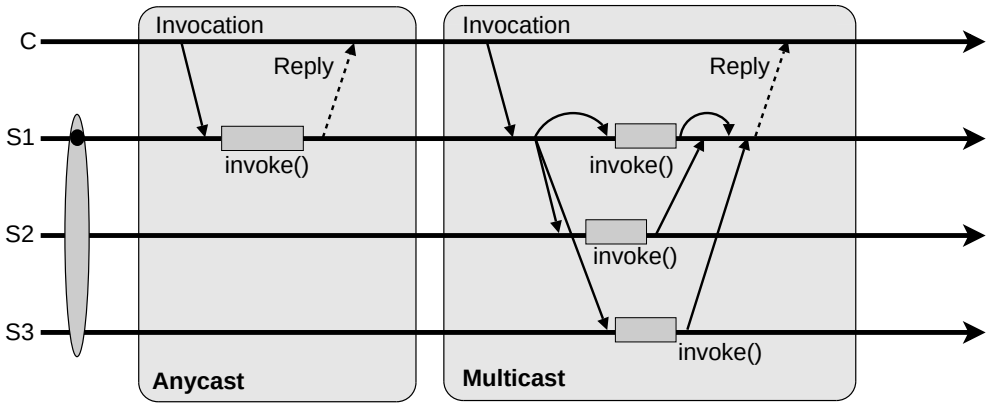


Figure 3.9: The supported EGMI method invocation semantics.

a single server. This server mediates the invocation to the other servers, as shown in Figure 3.6 above. Furthermore, the multicast invocation semantic does not order invocations and hence strong replica consistency cannot be enforced with this protocol. The atomic multicast protocol discussed in Section 7.5 fulfills these requirements.

Each of the methods exported to clients through the external interface have distinct invocation semantic, and the choice of semantic to associate with each method is left to the application developer designing the external interface. The default semantic for an external method is anycast, and methods with multicast semantic have to be tagged by including a specific exception in their *throws* clause.

Note that this is improper use of the exception declaration mechanism. Therefore, in recent versions of Jgroup this mechanism has been replaced with a more suitable mechanism based on Java annotations [6, Ch.15] as discussed in Chapter 7.

Our implementation of Jgroup guarantees that EGMI are *live*: if at least one server remains operational and in the same partition as the invoking client, EGMI will eventually complete with a reply value being returned to the client. Furthermore, an EGMI is completed by each server *at-most-once*, and only if some client has previously performed it. These properties hold for both anycast and multicast versions of EGMI. In the case of multicast EGMI, Jgroup also guarantees view synchrony as defined in the previous section.

Internal and external GMI differ in one important aspect. Whereas an IGMI, if it completes, is guaranteed to complete in the same view on all servers, an EGMI may complete in several different concurrent views. This is possible, for example, when a

server completes the EGMI but becomes partitioned from the client before delivering the result. Failing to receive a response for the EGMI, the client's group proxy has to contact other servers that may be available, and this may cause the same EGMI to be completed by different servers in several concurrent views. The only solution to this problem would be to have the client join the group before issuing the EGMI. In this manner, the client would participate in the view agreement protocol and could delay the installation of a new view in order to guarantee the completion of a method in a particular view. Clearly, such a solution may become too costly as group sizes would no longer be determined by the number of servers (degree of replication), but by the number of clients, which could be very large.

One of the goals of Jgroup has been the complete transparency of server replication to clients. This requires that from a client's perspective, EGMI should be indistinguishable from standard Java RMI. This has ruled out consideration of alternative definitions for EGMI including multi-value results or asynchronous invocations. Note that the client-side group proxy may still receive multiple result values, but these cannot be exposed to the client application.

3.3.3 The State Merging Service

While partition-awareness is necessary for rendering services more available in partitionable environments, it can also be a source of significant complexity for application development. This is simply a consequence of the intrinsic availability-consistency tradeoff for distributed applications and is independent of any of the design choices we have made for Jgroup.

Being based on a PGMS, Jgroup allows partition-aware applications, which have to cope with multiple concurrent views. Application semantics dictate which of its services remain available where during partitioning. When failures are repaired and multiple partitions merge, a new common server state has to be constructed. This new state should reconcile, to the extent possible, any divergence that may have taken place during partitioned operation.

Generically, state reconciliation tries to construct a new state that reflects the effects of all non-conflicting concurrent updates and detect if there have been any conflicting concurrent updates to the state. While it is impossible to automate completely state reconciliation for arbitrary applications, a lot can be accomplished at the system level to simplify the task [13]. Jgroup includes a state merging service (SMS) that provides support for building application-specific reconciliation protocols based

on stylized interactions. The basic paradigm is that of full information exchange – when multiple partitions merge into a new one, a coordinator is elected among the servers in each of the merging partitions; each coordinator acts on behalf of its partition and diffuses state information necessary to update those servers that were not in its partition. When a server receives such information from a coordinator, it applies it to its local copy of the state. This one-round distribution scheme has proven to be extremely useful when developing partition-aware applications [15, 86].

Figure 3.10 illustrates two runs of the state merge algorithm. The first is failure-free; $S1$ and $S4$ are elected as coordinators for their respective partitions, and successfully transfer their state. The second case shows the behavior of the state merge in the event of a coordinator crash ($S4$). In this case, the PGMS will detect the crash, and eventually install a new view. This will be detected by the SMS, that will elect a new coordinator for the new partition, and finally complete the state merge algorithm.

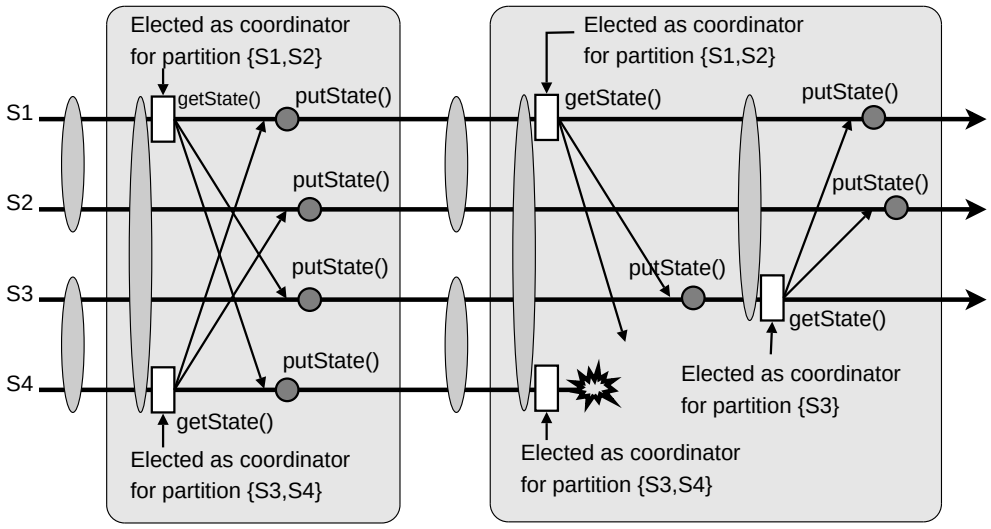


Figure 3.10: Two runs of the state merge algorithm: (i) two partitions merge and no failure occurs; (ii) two partitions merge and a coordinator fails.

SMS drives the state reconciliation protocol by calling back to servers for "getting" and "merging" information about their state. It also handles coordinator election and information diffusion. To be able to use SMS for building reconciliation protocols, servers of partition-aware applications must satisfy the following requirements: (i) each server must be able to act as a coordinator; in other words, every server has to

maintain the entire replicated state and be able to provide state information when requested by SMS; (ii) a server must be able to apply any incoming updates to its local state. These assumptions restrict the applicability of SMS. For example, applications with high-consistency requirements may not be able to apply conflicting updates to the same record. Note, however, that this is intrinsic to partition-awareness, and is not a limitation of SMS.

The complete specification of SMS is given in [87]. Here we very briefly outline its basic properties. The main requirement satisfied by SMS is *liveness*: if there is a time after which two servers install only views including each other, then eventually each of them will become up-to-date with respect to the other, either directly or indirectly through different servers that may be elected coordinators and provide information on behalf of one of the two servers. Another important property is *agreement*: servers that install the same pair of views in the same order are guaranteed to receive the same state information through invocations of their "merging" methods in the period occurring between the installations of the two views. This property is similar to view synchrony, and like view synchrony may be used to maintain information about the updates applied by other servers. Finally, SMS satisfies *integrity*: it will not initiate a state reconciliation protocol without reason, e.g. if all servers are already up-to-date.

3.4 The Dependable Registry Service

As previously discussed, when a client wants to communicate with a Java RMI server (or an object group), it needs to obtain a reference to either the single server or the object group depending on what kind of server the client is trying to access. In the case of Java RMI, the Java runtime environment is bundled with a standard registry service called rmiregistry. This registry service is a simple repository facility that allows servers to advertise their availability, and for clients to retrieve references for remote objects. Unfortunately, the rmiregistry is not suitable for the distributed object group model. There are several reasons for this incompatibility:

1. The rmiregistry constitutes a single point of failure. It is commonly used for bootstrapping clients with stubs for servers, and the failure of the registry will render *new* clients unable to establish a connection with the server (through Java RMI, that is). The failure of the rmiregistry have no impact on existing client-server communication, unless a client has an obsolete stub reference.

2. It does not support binding multiple server references to a single service name, as required by a group communication system.
3. Only local servers are allowed to bind their stubs in the rmiregistry. This limitation precludes replication, as a group of servers will necessarily involve servers running on distinct (non-local) hosts.

To address these incompatibilities, Jgroup includes a *dependable registry* (DR) service [86, 87], that can be used by servers to advertise their ability to provide a given service identified by the service name. In addition, clients will be able to retrieve a group proxy for a given service, allowing them to perform method invocations on the group of servers providing that service, as discussed in Section 3.3.2. The DR is designed as a replacement for the rmiregistry, and only minor modifications are required on both client and server side to adopt the DR.

The DR is in essence an actively replicated database, preventing the registry from becoming a single point of failure. It is replicated using Jgroup itself, and clients access the registry through EGMI. The database maintains mappings from service name sn to the set of servers $\mathcal{G}^{sn}$ providing that particular service. Thus, each entry in the database can be depicted as follows:

$$sn \rightarrow \mathcal{G}^{sn} \subseteq \{S_1, S_2, \ldots, S_n\}$$

where S_i denotes a server, and n represents the number of servers registered under the service name sn in the DR.

The set of servers $\mathcal{G}$ should always contain at least the active servers, i.e. those that are in the current view $\mathcal{V}$. Hence, a server must bind its reference with DR as soon as it starts, as illustrated in Figure 3.4. This allows clients to query the DR to obtain $\mathcal{G}$ to communicate with the group.

However, $\mathcal{G}$ may contain a substantial number of stale servers, if the DR is not updated in any way. This would cause clients to attempt invocations to stale servers (before removing them), which introduces a significant failover latency. Therefore, $\mathcal{G} = \mathcal{V}$ is desirable, which can be achieved by updating $\mathcal{G}$ according to $\mathcal{V}$ in the current partition as discussed in [78] and Chapter 9.

Chapter 4

An Overview of Autonomous Replication Management

As discussed in Chapter 3, Jgroup solves a number of difficulties involved in developing dependable distributed applications. However, the application developer still has to consider a number of complicated issues, e.g. those related to fault treatment, such as having to deal with replica recovery and deployment to best satisfy the dependability requirements of the application.

The aim of the *Autonomous Replication Management* (ARM) framework [77, 80, 81] is to provide generic services and mechanisms that can assist applications in meeting their dependability requirements. To further simplify the deployment and operation of dependable applications, ARM also seeks to reduce the required human interactions through a self-managing fault treatment architecture that is adaptive to network dynamics and changing requirements.

ARM extends Jgroup with automated mechanisms for performing fault treatment and management activities such as distributing replicas on sites and nodes, and recovering from failures thus reducing the need for human intervention. These mechanisms are essential to operate a system with strict dependability requirements, and are largely missing from existing group communication systems [87, 43, 18, 3].

Being based on Jgroup, ARM inherits the same system model as described in Section 3.1, while ARM enhances the architectural model of Jgroup by introducing object factories, configurable protocol modules, and a distributed service for managing deployed services.

This chapter gives a brief overview of the services and mechanisms provided by the ARM framework, whereas Chapter 11 provides additional details. Section 4.1 presents a short description of the mechanisms embedded within ARM, relating them to autonomic computing concepts. In Section 4.2 an overview of Jgroup/ARM is presented, aimed at illustrating the high-level relations between ARM and Jgroup. Finally, in Section 4.3 a brief introduction to the ARM architecture is given.

4.1 Introduction

To support its goals, the ARM framework is comprised of three core mechanisms: *policy-based management*, *self-healing* and *self-configuration* as discussed below.

Policy-based management [113], where application-specific replication policies and a system-wide distribution policy are used to enforce the dependability requirements and WAN partition robustness of services. Policy-based management allows a system administrator to provide a high-level policy specification to guide the behavior of the underlying system. Three distinct policies are used to guide ARM in making decisions on how to handle a particular system condition, e.g. a replica failure. The system-wide *distribution policy* is specific to each ARM deployment, and is used to determine the set of sites and nodes (see Figure 1.1 on page 8) on which replicas can be allocated. A *replication policy* and a *remove policy*, both application-specific, are used guide the behavior of ARM with respect to maintaining the dependability requirements and recovery needs of an application service.

Self-healing [95], where failure scenarios are discovered, diagnosed and handled through recovery actions, depending on the replication policy of the affected services. The objective of this mechanism is to minimize the period of reduced failure resilience, in which additional failures could potentially cause the service to stop, ultimately degrading its dependability characteristics. Currently, *object*, *node* and *network partition* failures are handled. It could be extended to handle value faults as well, like Delta-4 [102] and AQuA [104]. However, that requires an interaction model where clients can obtain results directly from all replicas, which is currently not supported by Jgroup. This was briefly discussed in Section 2.4.1. ARM is also able to handle multiple concurrent failure activities of the same or different services, including failures affecting the ARM infrastructure (self-recovery). The latter requires that core parts of the ARM infrastructure is replicated to avoid becoming a single point of failure.

Self-configuration [95] is supported by adapting to changes in the environment, allowing service replicas to be relocated/removed to adapt to uncontrolled changes such as failure/merge scenarios, or controlled changes such as scheduled maintenance. For example, operating system upgrades can be performed without manual intervention to migrate replicas to alternative locations. A node is simply removed from the system, causing ARM to reconfigure the replica locations, and upon completion of the maintenance activity the node is reinserted. ARM also supports software upgrade management [115, 114], as discussed in Chapter 12.

The ARM framework could also be extended with some degree of self-optimization in the sense that policies can be programmed to determine the optimal redundancy level for the current network environment.

A non-intrusive system design is applied, where the operation of deployed services is completely decoupled from ARM during normal operation in serving clients. Once a service has been installed, it becomes an "autonomous" entity, monitored by ARM until explicitly removed. This design principle is essential to support a scalable architecture, and follows naturally from the open group model [64] adopted by Jgroup. Only a negligible overhead is added to each deployed service, due to the main recovery mechanism provided by ARM. Additional recovery mechanisms can be added at the cost of additional overhead.

4.2 Jgroup/ARM Overview

This section briefly describes the interactions between the various Jgroup and ARM components. Figure 4.1 shows a high-level view of the components and communication patterns of Jgroup/ARM.

A client interacts with the dependable registry to obtain group proxies for services that the client needs to perform its operations. Prior to this, the servers must bind their references in the registry. Servers will also notify ARM of changes to the membership of their respective groups. These notifications are used by ARM to determine if recovery is needed.

The various servers are mapped to nodes (and sites) in the target environment (see Figure 1.1). Each node has a factory which is used to install and remove servers on that node. ARM is also responsible for installing and removing services on demand. Figure 4.2 shows the installation of a triplicated server group using the ARM framework.

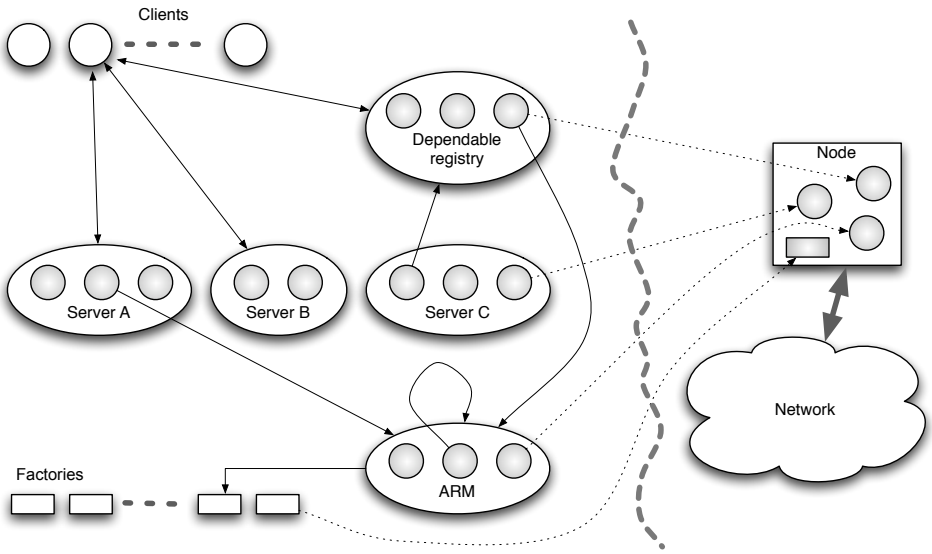


Figure 4.1: Overview of Jgroup/ARM components and communication patterns.

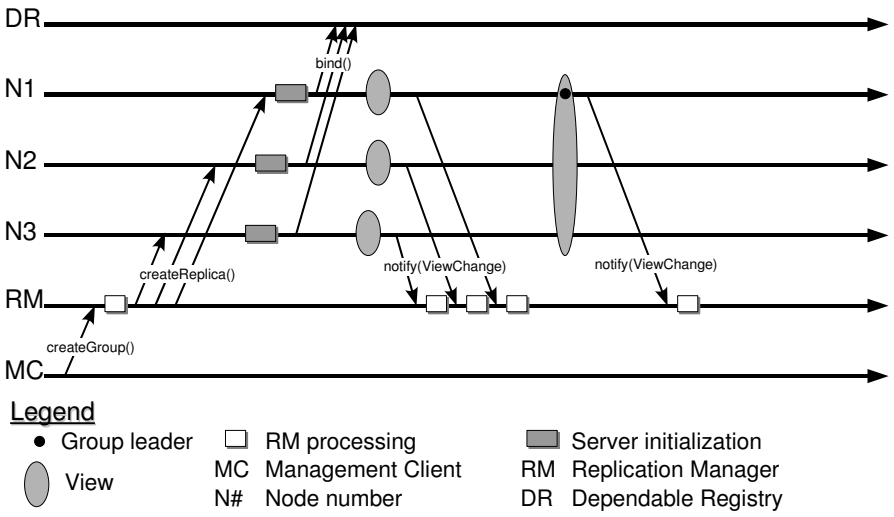


Figure 4.2: Installation of a triplicated server group.

The responsibilities of the ARM framework include deployment and operation of dependable applications "seamlessly" within a predefined *target environment*, i.e. the set of sites and nodes that may host applications and ARM-specific services. Within the bounds of the target environment, ARM autonomically manages both replica distribution, according to the rules of the distribution policy, and recovery from various failure scenarios, based on the replication policy of the service(s) exposed to the failure. For example, maintaining a fixed redundancy level is a typical requirement specified in the replication policy. An example of a typical failure-recovery sequence is shown in Figure 4.3, in which node $N1$ fails, followed by a recovery action causing ARM to install a replacement replica at node $N4$.

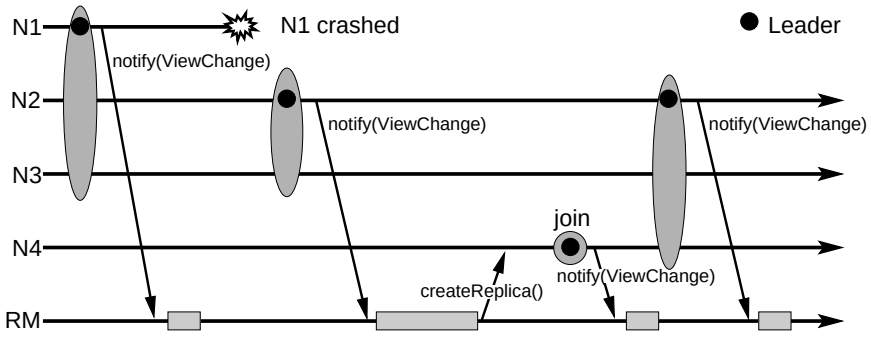


Figure 4.3: An example failure-recovery sequence.

4.3 Architectural Overview

The ARM framework keeps track of deployed services in order to discover failures and to initiate recovery to compensate for reduced failure resilience. This section gives a high-level overview of the ARM architecture.

Figure 4.4 illustrates the core components and interfaces supported by the ARM framework: a system-wide *replication manager* (RM), a supervision module associated with each of the managed replicas, an object factory deployed at each of the nodes in the target environment, and an external *management client* (MC) used to interact with the RM.

The *replication manager* is the main component of ARM; it is implemented as a distributed service replicated using Jgroup. Its task is to keep track of deployed services

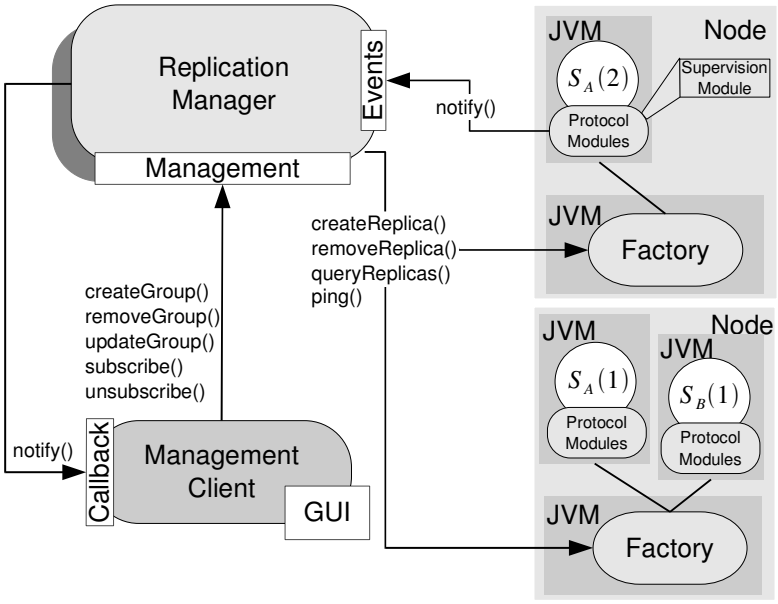


Figure 4.4: Overview of ARM components and interfaces.

in order to collect failure information, analyze this information, and reconfigure the system on-demand according to the configured policies. Reconfiguration often entails the creation of additional replicas to substitute crashed or partitioned ones, or the removal of excess replicas that may have been created due to incorrect failure suspicions or when partitions merge after repairs.

The *supervision module* is the ARM agent co-located with each Jgroup replica. It is responsible for forwarding view change events generated by the PGMS to the RM. It is also responsible for decentralized removal of excess replicas. The supervision module is one of several protocol modules associated with each replica. Protocol modules are discussed in Chapter 6.

The purpose of *object factories* is mainly to act as bootstrap agents; they enable the RM to install and remove replicas, as well as to respond to queries about which replicas are hosted on the node. They also provide mechanisms to assist ARM in making decisions about the nodes on which to place replicas.

The *management client* provides system administrators with a management interface, enabling on demand installation and removal of dependable applications in the target environment and to specify and update the distribution and replication policies to be

used. The management client can also acquire information about running services through callbacks from the RM.

Note that it is assumed that the RM, along with the managed services (such as the S_A and S_B services in Figure 4.4), is deployed within the same target environment, while the management client and other clients are considered external to the target environment.

Overall, the interactions among these components enable the RM to make proper recovery decisions and allocate replicas to suitable nodes in the target environment. These tasks are performed by ARM without human interaction.

Part II

Adaptive Middleware

Chapter 5

The Jgroup/ARM Architecture

To support fault treatment, the underlying architecture must support adaptive reconfiguration of deployed services. However, several design choices were made for Jgroup that are unfavorable with respect to support such reconfigurations.

Hence the purpose of this chapter is to present a common Jgroup/ARM architecture that satisfies the requirements needed to meet our goals. The architecture is based on the Jgroup model presented in Chapter 3, but a number of enhancements (Part II) and additions (Part III) have been made to address the fault treatment issue. Several advanced mechanisms are provided to support deployment and operation of self-managed distributed services. Below a brief overview of Part II is given.

The first enhancement is the ability to dynamically configure the group manager (see Section 3.2) according to some application-specific policy. The group manager can now easily incorporate new group-related services, in addition to the basic Jgroup services (PGMS, GMIS, SMS). This change makes it possible to build generic Jgroup services that can intercept group-related events and perform various tasks, e.g. reconfigurations, based on these events. This enhancement is covered in Chapter 6.

The second enhancement deals with supporting different replication protocols for EGMI. To build fault tolerant systems with strong consistency, simple anycast and multicast are not sufficient. Hence the EGMI part of Jgroup has been completely revised with an architecture for per method selection of replication protocols. Two new replication protocols have also been added; *atomic multicast* and *leadercast*. The latter is a variant of passive replication. Chapter 7 covers this enhancement.

Enhancements have also been made to the Jgroup daemon to support multiple replicas per node, by allowing several group managers to connect to the same daemon. This

change is essential, assuming that the number of nodes in the pool are less than the total number of server replicas to be deployed, promoting the reuse of node resources. Chapter 8 elaborates on the resource sharing enhancements.

The last enhancement, covered in Chapter 9, concerns the handling of group membership information on the client-side. Inconsistent client-side membership information can lead to performance penalties for anycast invocations, and also longer failover latency.

In the following section, the requirements for the Jgroup/ARM architecture is presented, supplementing the Jgroup model. In Section 5.2 the architecture is presented; the aim is to relate the various Jgroup and ARM infrastructure components into a common architecture.

5.1 Architectural Requirements

Given the goals, constraints and assumptions in Section 1.2.1, the following list of high-level requirements for the Jgroup/ARM architecture is derived. The architecture must support:

1. Deploying replicas onto nodes.
2. Running multiple replicas per node.
3. Failure independence between replicas on the same node.

The architecture is specifically designed to support the above requirements, but is also flexible enough to support alternative configurations.

To support requirement 1 in the list above, each node must provide some entity through which replicas can be deployed. One solution to this could be to use a program for remote execution, e.g. secure shell (`ssh`) [109]. However, such solutions are often specific to the operating system. Our solution is independent of operating system.

Requirement 2 in the list allows each node to host multiple replicas, thereby reusing resources. In general there are two approaches to hosting multiple replicas per node:

1. Each replica on the node communicates through separate ports and
2. all replicas on the node communicates through the same port.

The advantage of the former approach is that each replica is completely independent of all other replicas on the node, whereas the latter approach have a common point of failure. In the architecture presented below the second approach is used; the reason for this is twofold: (i) it is more scalable as communication resources can be shared between replicas (see Chapter 8) and (ii) it is easier to manage communication port allocations. This approach is also used by the Spread toolkit [3].

Though sharing the communication endpoint between replicas on the same node, failure independence may still be preserved for the replicas as given by requirement 3, as long as the communication endpoint component remains available. To ensure replica failure independence, the replicas must be running in separate processes (or JVMs). All replicas running in a shared JVM are likely to fail if one of the replicas fails.

5.2 The Jgroup/ARM Architecture

Figure 5.1 illustrates the principal components of Jgroup/ARM. The target environment is comprised of a set of server nodes; the server nodes may reside at separate sites in the network, albeit not shown explicitly in the figure. Client nodes are not part of the target environment; however, the client applications needs to be aware of the sites and nodes in the target environment to be able to find services.

In the figure, two application services have been deployed, S_A and S_B. S_A has three replicas, where replica number i is denoted $S_A(i)$, and S_B has two replicas. The replicas are distributed evenly over the four server nodes, such that the same node have at most one instance of each application service type. These two applications are deployed and operated through the ARM infrastructure discussed below. The figure also shows clients, C_A and C_B, for the corresponding server applications. Communication between a client and an object group is mediated by a group proxy (GP), that transmit group method invocations to the group managers associated with replicas forming the group.

A number of infrastructure components are also shown in Figure 5.1. These components perform various tasks to ensure that the dependability requirements of applications are maintained, and makes the development of such applications easier by hiding the non-functional details that can be handled by the framework instead of the application. The following gives a brief overview of these components and their purposes, and how the various components relate to each other.

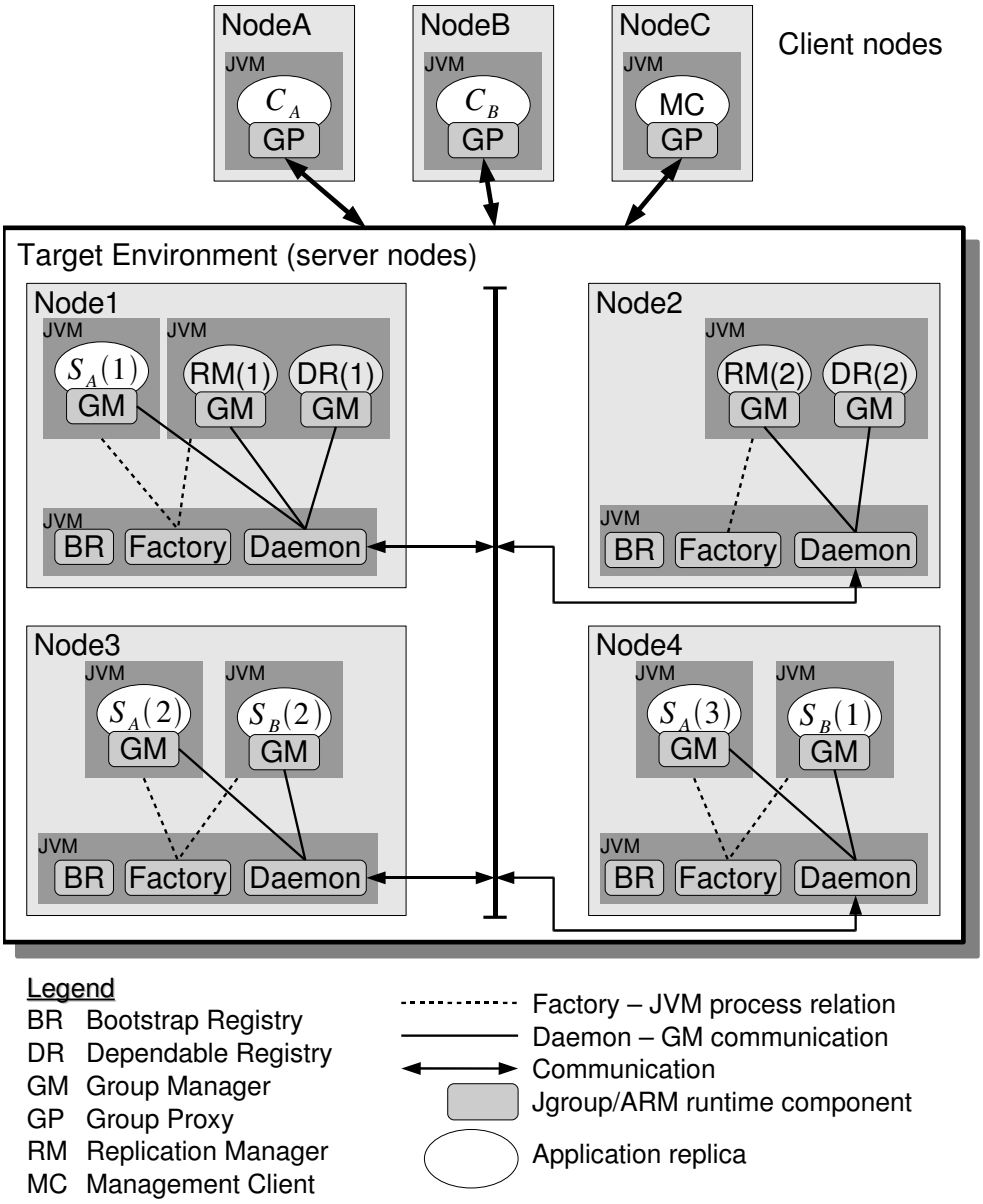


Figure 5.1: Architectural overview of Jgroup/ARM components.

The main components of the ARM infrastructure are described briefly in Section 4.3. These descriptions suffice for now, as we try to relate the various Jgroup and ARM components into a common architecture in the following sections.

5.2.1 Replication Manager Dependency

The replication manager (RM) relies on the dependable registry (DR) service to store its object group reference, enabling RM clients such as the supervision module, factory and management client to query the DR to obtain the group reference of the RM. Due to this dependency, ARM has been configured to co-locate RM and DR replicas in the same JVM, as illustrated in Figure 5.1. This eliminates the possibility that partitions separate RM and DR replicas, which could potentially prevent the system from making progress.

5.2.2 The Object Factory

Each node in the target environment must be running an object factory to be able to host server replicas. As illustrated with the dashed lines in the figure, the factory keeps track of its locally installed replicas at the JVM (process) level. Although the factory exists on each node, it does not form a group and only maintains local state information about replicas. The RM and other ARM components communicate with the factory through RMI.

The factory is co-located with a *bootstrap registry* (BR). The purpose of the BR is to store the remote reference of certain Jgroup/ARM infrastructure components, allowing other components to find them and enable communication required to bootstrap the system. The BR is simply a standard *non-replicated* Java RMI [123] registry service running on a *well-known port* on each node within the target environment. Each node contains only a single instance of the BR. The BR only keeps information about Jgroup/ARM components on the local node, such as the remote reference of the local factory, and if present the local dependable registry instance. The factory itself (and consequently the bootstrap registry) is typically started during the operating system boot process.

5.2.3 The Group Manager

Each replica is associated with one *group manager* (GM) for each group that the replica has joined[1]. The group manager represents the Jgroup runtime and is composed of a set of *protocol modules* implementing the Jgroup services described in Chapter 3, in addition to other group-related services such as message multicasting and supervision. Protocol modules may interact with (i) other local modules or the application, (ii) a corresponding remote module within the same group, and (iii) external entities such as clients. Local interactions, such as the EGMI cooperating with the PGMS to enforce view synchrony, are governed through internal service interfaces; each module *provides* a set of services, and *requires* a set of services to work. The set of protocol modules is dynamically constructed at runtime based on a declarative specification given at deployment time, as will be discussed in Chapter 6. This allows maximum flexibility in activating required services. The module configuration is integrated into the ARM policy management, so its description is deferred to Chapter 10.

As shown in Figure 5.1, multiple GMs may reside on the same node, either in the same JVM or in different JVMs. Typically, distinct application replicas residing in the same JVM belong to different groups, exemplified by the RM and DR replicas. As discussed above, such co-location can be favorable if there is a dependency between the applications.

5.2.4 The Jgroup Daemon

The basic group communication facilities provided by Jgroup are partially implemented in the *daemon*. However, the servers do not interact with the daemon directly, but rather through the GM. The GM of each replica must connect to a daemon. The new daemon architecture (discussed in Chapter 8) is able to handle multiple GM connections, reducing the cost of group communication to the number of daemons, and being able to reuse nodes by co-locating multiple replicas on the same node. Note that the daemon is only used for low-level group-related communication among group members, whereas client and server implementations interact through high-level interfaces based on GMI. As illustrated in Figure 5.1 multiple replicas connect to the node local daemon. Communication between the daemon and its connected

[1] A server may join multiple groups, but in this work we consider only servers (replicas) joining a single group.

GMs take the form of remote method invocations. It is also possible for a GM to connect to a daemon on a remote node (not shown in the figure). Replicas belonging to different applications may share the same daemon, but it is also possible to run several daemons on the same node when distinct applications have completely different membership and communication requirements.

5.2.5 Failure Independence and JVM Allocation

The architecture in Figure 5.1 is flexible in that application replicas and various Jgroup/ARM runtime components may be located in separate JVMs. This is motivated by the desire to enhance the failure independence of different Jgroup/ARM components. In particular, each application replica is executed in a different JVM so as to prevent misbehaving replicas from disrupting the Jgroup/ARM runtime or other applications located on the same node. However, failure independence comes at the cost of interprocess communication (RMI), and for certain applications this additional cost may be too high. To avoid the interprocess communication overhead, it is possible to configure application replicas to be co-located in the same JVM as the Jgroup/ARM runtime, sacrificing failure independence. This case is not shown in Figure 5.1, but is discussed further in Chapter 8.

Chapter 6

Dynamic Protocol Composition

Without proper support for configurable protocol stacks, the application developer needs to have detailed knowledge of the core group communication toolkit, leading the developer to worry about issues outside of the application domain. Moreover, the developer would need to recompile from the source code for each protocol stack implemented, which would again lead to issues of maintaining multiple versions of the code. In the first implementations of Jgroup, adding alternative protocols or group-related services required modifying core components of the toolkit. Jgroup lacked flexibility in several aspects, including the composition of protocol modules.

To equip Jgroup with a framework for adaptive and configurable protocol composition, core parts of the Jgroup toolkit had to be redesigned. The changes discussed in this chapter makes it possible for each individual application to dynamically configure its own group communication protocol stack, and each protocol module can be parameterized according to application-specific requirements.

This chapter is organized as follows: In Section 6.1 we discuss previous works on protocol architectures and relate these to the approach taken by Jgroup. Section 6.2 states the requirements for our protocol architecture. Section 6.3 introduces the concepts on which our protocol composition framework is based, and illustrate a sample protocol stack. Section 6.4 details the various ways in which protocol modules can communicate, both internally and externally. Finally, in Section 6.5 the dynamic composition of protocol modules is discussed.

6.1 Introduction to Protocol Architectures

Protocol composition is traditionally based on ISO/OSI like protocol stacks. However, in the last decade, micro-protocols have become increasingly popular, as they are aimed at providing more flexible ways for protocol composition. To accomplish this, micro-protocol frameworks restrict their protocol layers to follow a specific model, rather than building protocols in an ad hoc manner. Examples of such restrictions include things like: the protocol layers have to communicate using events that travel up or down the protocol stack, and that the layers cannot share any state. This way protocols become more maintainable and configurable as new protocols can easily be added to the system. The cost however, is reduced performance.

Micro-protocols were first introduced in the *x*-kernel [60], and have since been used in a variety of systems, including group communication systems such as Ensemble [55], Horus [132], JavaGroups [18], Cactus [58] and Appia [84]. Jgroup does not use a micro-protocol architecture in the low-level message communication (see Chapter 8). However, the group manager architecture discussed in this chapter, can be compared to micro-protocol architectures.

Ensemble, Horus, JavaGroups and Appia [55, 132, 18, 84] follow a strictly vertical stack composition, where events must pass through all layers in the stack. In the Horus system, a protocol accelerator [131] implements optimizations that reduce the effects of protocol layering. The limitation of these optimization techniques is that the set of protocols to be bypassed must be well-defined, and the optimizations were hand-coded into the protocol stack. Thus, it reduces the configurability of the micro-protocol framework. Similar optimizations are also feasible with the Ensemble system [55]. Both Appia [84] and JavaGroups [18] are also based on micro-protocols in its purest form, since none of the optimizations implemented in Horus and Ensemble are available. That means that every event has to pass through all intermediate layers, even though the event is not being processed by all of the layers.

The Cactus [58] micro-protocol framework is conceptually similar to the Jgroup protocol composition framework discussed in the remainder of this chapter. Each layer has to register its interest in the events of other layers, and protocols can be constructed according to formal rules, such as a dependency graph. Thus, such a protocol stack does not follow a strict vertical composition. An advantage of the Jgroup protocol framework over JavaGroups, Appia and the Cactus system is type-safety. Events are passed by means of method calls on a set well-defined interfaces for the various modules (layers), whereas other systems have to implement a common handler

method in each layer which takes care of demultiplexing the received events based on the type of the events. In Jgroup, events are passed directly to the appropriate event handler. Another advantage of Jgroup over the Cactus system is the possibility to specify interception rules, enabling a module to delay and/or modify events from another module.

6.2 Protocol Architecture Requirements

In the original Jgroup implementation, protocol modules were constructed statically as a monolithic core, and message passing relations between the various protocol modules were established in an ad hoc manner. Hence, the requirements for redesigning the protocol framework for Jgroup include:

1. Support for simple inter- and intra stack message passing.
2. Efficient intra-stack message passing.
3. Dynamic construction of protocol stacks.

The first requirement above was already provided through an ad hoc mechanism for connecting protocol modules. Message passing between different stacks was also supported, but also here ad hoc mechanisms were used and there was a lack of support for multiplexing messages destined for different modules.

The main contribution for the redesigned protocol architecture is in the dynamic construction of protocol stacks. However, also the message passing techniques used in Jgroup have been redesigned to follow certain rules that make it easier to develop protocol modules, and to check the structural correctness of a given set of protocol modules.

6.3 Protocol Modules

The Jgroup *group manager* (GM) is the glue between an application and the core group communication services. It allows the application to interface with the various Jgroup services to perform group-specific tasks. The GM is based on an *event-driven non-hierarchical composition model*[1], and consists of a set of *weakly coupled protocol modules*. Each protocol module implements a group-specific function, which *may*

[1] In [82] this is called *cooperative composition*.

require the collaboration of all group members, e.g. the membership service (PGMS). In fact, all the basic Jgroup services discussed in Chapter 3 and several other generic group-specific functions are implemented as GM protocol modules.

The advantages of a non-hierarchical set of protocol modules over a strictly vertically layered architecture, as used in many other group communication systems (e.g. [132, 55, 18, 84]), is that events being passed from one layer (module) does not have to be processed by any intermediate layers. Events can simply be passed from one module to another without any processing delay and addition/removal of header fields, thus also reducing the complexity of implementing a module. Our approach is also flexible in that a module can intercept commands/events from another module, delay and/or modify them, before delivery to the destination module. Interception rules are specified inline in the modules using Java annotations, and the corresponding implementations must adhere to these rules.

Protocol modules communicate with the application, or other modules, by means of *commands* (downcalls) and *events* (upcalls) through a set of well-defined interfaces. Typically, a module *provides* a set of services to other modules and/or the application, and *requires* another set of services from other modules to perform its services. A module may also *substitute* the services provided by another module, by intercepting, delaying and/or modifying the commands/events passed on to the substituted module.

Each module implements one or more well-defined *service interfaces*, through which the module can be controlled, and it may also generate events to listening modules (or the application) through one or more *listener interfaces*. Usually, a module implements one service interface and provide events to other modules through one listener interface. As an example, consider the MembershipModule which is defined by the MembershipService and MembershipListener interfaces, shown in Figure 6.1. Notice also that the server replica may use the MembershipService interface to join/leave the group. To facilitate this, access to the service interfaces of protocol modules can be obtained through a static method on the group manager class. However, to be notified of events generated by the various modules, a server only needs to implement the listener interface(s) of the module.

The set of GM protocol modules required by an application is configured through the ARM policy management discussed in Chapter 10. Based on this configuration, the protocol modules are constructed dynamically at runtime. Recent versions of JavaGroups [18] and Appia [84] also supports construction of protocol stacks based on a configuration file. There is no strict ordering in which the modules have to be constructed, except that the set of required modules must have been constructed

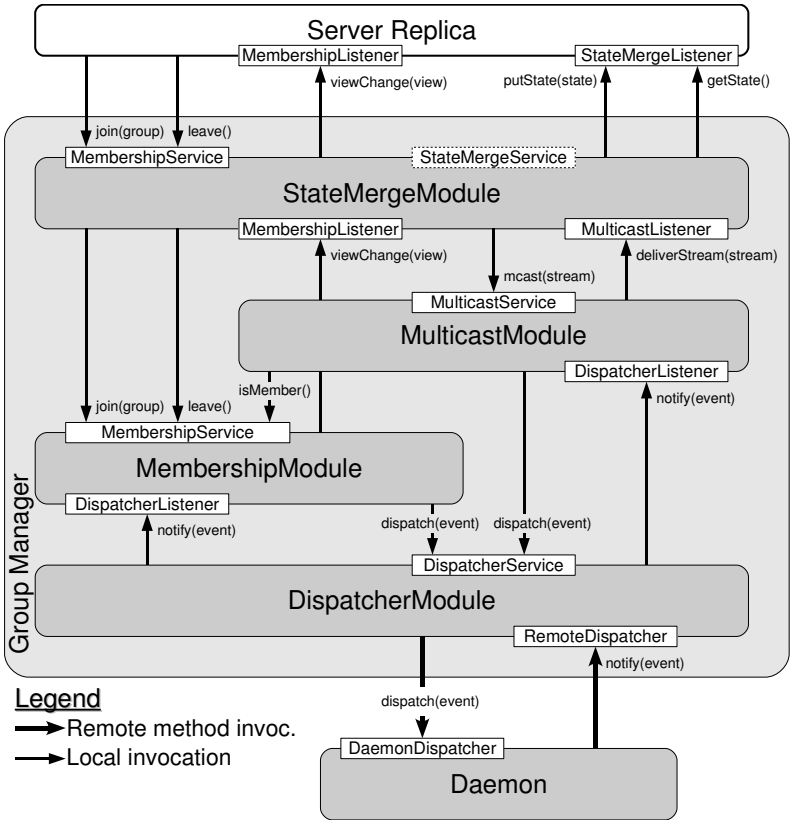


Figure 6.1: A sample group manager composition with the basic Jgroup services.

a priori. During construction, each module is checked for structural correctness, and required modules are constructed on-demand.

Constructing protocol modules dynamically has the advantage of enabling developers to easily build generic group-specific functions and augment the system with new modules without having to recompile the complete framework.

Figure 6.1 illustrates a protocol composition containing the basic Jgroup services, except the GMIS. For readability only the most important commands/events are shown in the interfaces.

The DispatcherModule is responsible for queuing and dispatching events to/from the daemon, and is the interface between the GM protocol modules and the daemon. The MulticastModule implements the MulticastService through which other modules (and the application) can send multicast messages to the current group members.

To receive multicast messages, a module must implement the MulticastListener interface. The main task of the MulticastModule is to multiplex and demultiplex the multicast messages to/from the internal modules or the server replica. The actual low-level IP multicast is performed by the daemon. Other modules (or the application) can join() or leave() a group by invoking the MembershipService interface (see Listing 6.1), which is implemented by the MembershipModule. Variations in the group membership are reported through viewChange() events. Any number of modules, and the application, may register its interest in such events simply by implementing the MembershipListener interface (see Listing 6.2). The MembershipModule mainly keeps track of various state information and provides an administrative interface to the PGMS, whereas the view agreement protocol is implemented in the daemon (for details see [87]). The DispatcherModule, MulticastModule and MembershipModule are mandatory, and must always be included for any sensible group communication support.

Note that the StateMergeModule also implements the MembershipService interface, and provides events through the MembershipListener interface. This is since the StateMergeModule substitutes the membership service by intercepting and delaying the delivery of viewChange() events to the server replica until after the state has been merged. The main task of the StateMergeModule is to drive the state reconciliation protocol by calling getState() and putState() on the StateMergeListener interface to obtain and merge the state of server replicas. It also handles coordinator election and information diffusion. State reconciliation is only activated when needed, i.e. in response to viewChange() events generated by the MembershipModule. Hence, the StateMergeService interface (dashed box) does not provide commands as a means for activating it. As Figure 6.1 illustrates, the StateMergeModule requires both the MembershipModule and the MulticastModule, and substitutes the MembershipModule.

6.4 Module Interactions

Protocol modules may interact in a number of different ways, both with external entities and other protocol modules. Previously, Jgroup supported similar interaction styles to those presented in this section, except for the interaction style discussed in Section 6.4.4. However, the interaction links were established in a static and ad hoc manner. Hence, in this section we formalize the various interaction styles used between modules. Furthermore, to construct the protocol modules dynamically, it is

necessary to understand the ways in which the modules can interact so as to dynamically establish the necessary links between them.

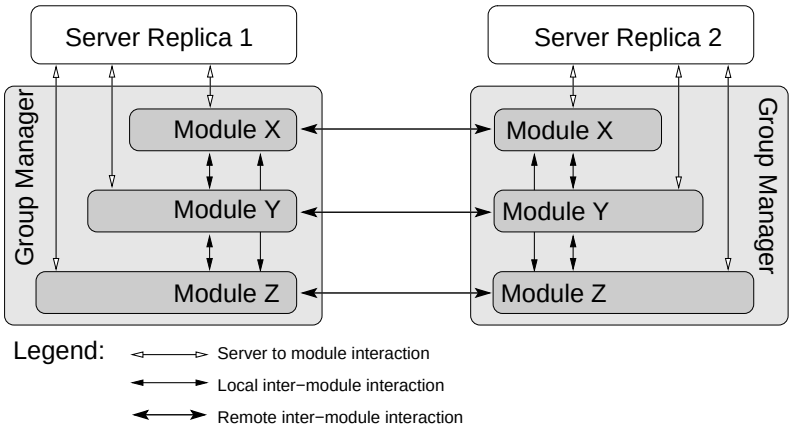


Figure 6.2: Inter-module and server-to-module interactions.

Figure 6.2 illustrates inter-module and server-to-module interactions. Inter-module interactions may occur both within the same GM, and also across distinct GMs. Mostly, only GMs that belong to the same group needs to communicate. GMs belonging to the same group should be composed of an identical set of protocol modules. The arrows in Figure 6.2 represents a *may* communicate relation. That is, a module may or may not communicate with another module in one or both directions. The server replicas may also communicate *directly* with one or more of the modules within its local GM, without passing through any intermediate modules. The thicker arrows represent remote communication between peer modules.
Four distinct forms of interaction styles involving protocol modules have been identified, as listed below. The first three are shown in Figure 6.2.

1. Local inter-module interactions between modules internal to the same GM.
2. Remote inter-module interactions between peer modules in distinct GMs.
3. Interactions between the server and local protocol modules.
4. Interactions between an external entity and a protocol module.

The last interaction style is common in many of the subsystems presented in later chapters. Basically, it allows a protocol module to notify or to be notified by an external entity. In the following, we discuss each of these interaction styles individually. Although commonplace, server-to-server interactions are not considered here.

6.4.1 Local Inter-module Interactions

As mentioned above, the GM is composed of a collection of protocol modules, each of which may provide a service to other modules in the same GM. In addition, a protocol module may also listen to events from other modules. Figure 6.3 illustrates a generic view of the internal inter-module interaction interfaces, through which local protocol modules communicate. In the figure, the service interface implemented by module A is used by modules B and C within the same GM to invoke commands offered through the service interface (e.g. to join() a group). Module A also implements a set of listener interfaces through which it can be notified of events generated by modules D and E (e.g. a viewChange() event.)

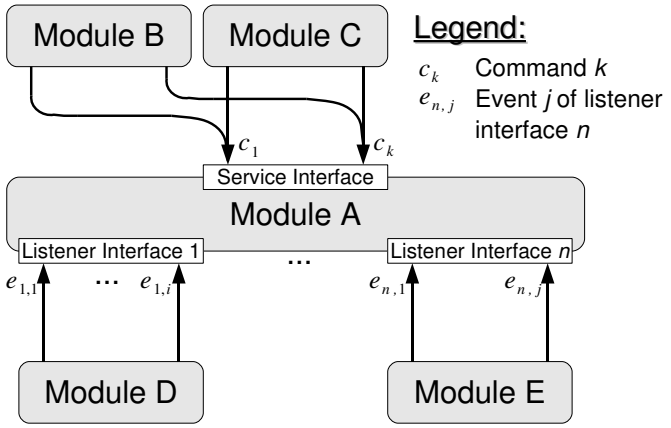


Figure 6.3: A generic view of the interfaces used for local inter-module interactions.

A module *must* implement at least one service interface, but may also implement more than one service (not shown in Figure 6.3). Implementing multiple service interfaces is useful when a module intercept and substitute the services of another module, e.g. the StateMergeModule in Figure 6.1. For most other circumstances a module should implement only a single service interface to encourage reuse.

The service interface typically contains one or more commands ($c_1, \ldots, c_k$), that can be invoked by the application or by other modules. The service interface may also be empty in the sense that it does not provide any commands (methods). Such empty interfaces are often called *marker* interfaces, and only serves the purpose of identifying the module internally in the GM. The dashed box around the StateMergeService interface in Figure 6.1 is used to indicate that it is an empty marker interface.

A module *may* have one or more associated listener interfaces through which module generated events can be passed to its listeners (other modules or the server replica). Figure 6.1 illustrates the use of multiple listener interfaces; the StateMergeModule generates events through both StateMergeListener and MembershipListener, since the StateMergeModule substitutes the MembershipModule. Usually however, a module generate events through a single listener interface (see Figure 6.3). A module without any associated listener interfaces is useful only when the module provides services commands.

A module *may* receive events ($e_{1,1}, \ldots, e_{1,i}$) generated by other modules by implementing one or more listener interfaces.

Listing 6.1: Partial view of the MembershipService interface.

```
1  public interface MembershipService
2    extends Service, ServiceFinalizer
3  {
4    public void join(int groupId)
5      throws JgroupException;
6    public void leave()
7      throws JgroupException;
8    public boolean isLeader();
9    public MemberId getMyIdentifier();
10   public MemberTable getMemberTable();
11 }
```

Listing 6.2: The MembershipListener interface.

```
public interface MembershipListener
{
  public void viewChange(View view);
  public void prepareChange();
  public void hasLeft();
}
```

The service and listener interfaces are defined in terms of Java interfaces, and arrows in Figure 6.3 represents Java methods (commands/events). Listings 6.1 and 6.2 illustrates two example interfaces, corresponding to the service and listener interfaces associated with the MembershipModule.

6.4.2 Remote Inter-module Interactions

Modules within one GM may interact with its remote peer modules in other GMs belonging to the same group. Previous versions of Jgroup only supported message multicasting at the module level, whereas server replicas could exploit IGMI. Therefore the InternalGMIModule has been redesigned by adding a routing mechanism to determine which of the modules or the server should receive a particular invocation. Hence, two approaches can now be used by module developers to support interaction between peer modules:

- Message multicasting (using the MulticastModule)
- Internal group method invocations (using the InternalGMIModule).

The advantage of the former approach is primarily efficiency, since it adds no overhead to the messages being sent by the module, except for a small header used to route multicast messages to the appropriate peer modules. The drawback with message multicasting is that module complexity increases, since the developer must implement marshalling and unmarshalling routines for the different message types to be exchanged between peer modules.

Contrarily, the InternalGMIModule takes care of marshalling and unmarshalling, reducing the module complexity to pure algorithmic considerations. The InternalGMIModule do however impose an additional overhead compared to that of message multicasting. The overhead is mostly due to the use of dynamically generated proxies [6, Ch.16], as discussed briefly below. Albeit not confirmed through measurements, we expect that the overhead imposed by the proxy mechanism is small compared to the communication latencies between the peer modules.

Figure 6.4 illustrates the workings of the InternalGMIModule as a means for communication between peer modules. The example shows the ExchangeModule introduced for the purpose of this discussion. It implements the InternalExchange interface, containing the exchange() method. This allows the ExchangeModule to invoke the exchange() method on its peer modules, including itself. To support such internal method invocations on peer modules, the InternalGMIModule dynamically constructs an IGMI proxy object implementing the same InternalExchange interface as the ExchangeModule. The IGMI proxy acts as a representative object for the whole group of peer modules. The ExchangeModule can then simply invoke the exchange() method on the proxy (❶), and the proxy will take care of marshalling the invocation and multicasting it to the other group members (❷). Then the InternalGMIModules

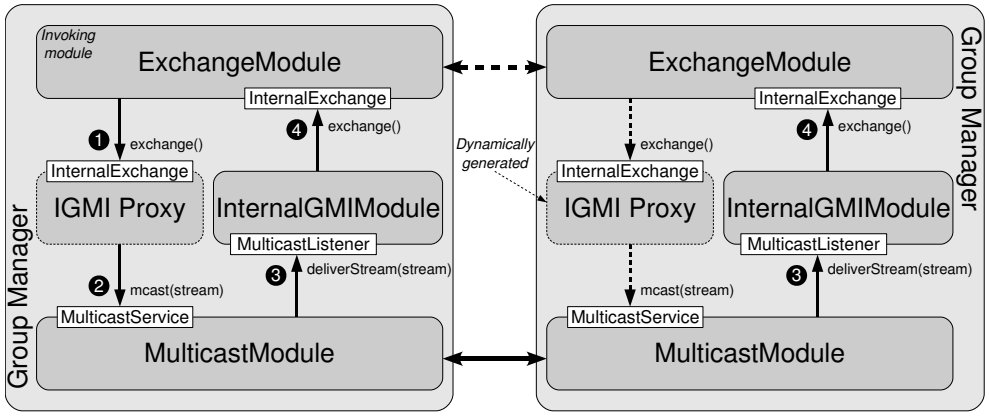


Figure 6.4: Remote inter-module interaction using the IGMI approach.

will receive the multicast message (❸), and will take care of unmarshalling and invoking the exchange() method on the local ExchangeModule (❹). Finally, the results are returned to the invoking module, following the reverse path (not shown in the figure).

6.4.3 Server to Module Interactions

As discussed above, also the server implementation may interact with the local modules. The original Jgroup implementation used the same interaction style as presented in this section; it is included here for completeness as it has not been documented elsewhere.

Figure 6.5 shows a generic view of the server-to-module interactions. A server replica *may* choose to listen to an arbitrary set of events generated by its associated protocol modules. To accomplish this, the server must implement the listener interfaces associated with the modules whose events are of interest. However, a server may also choose to not implement any listener interfaces if it does not need to process events generated by modules.

In a similar manner, the server *may* invoke any one of the commands provided through the service interfaces of the protocol modules associated with the server.

As Figure 6.5 demonstrates, various combinations of using services and listening to events are possible. The server may both listen to events of a module, and invoke its

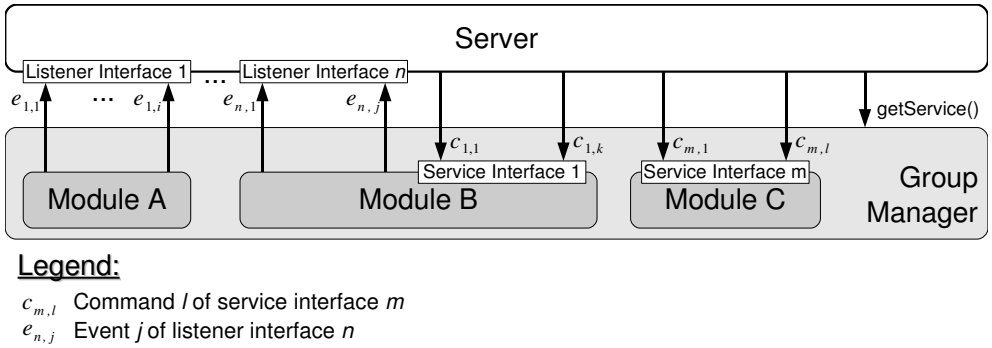


Figure 6.5: A generic view of the server-to-module interaction interfaces.

service commands (middle module), or it may just listen to its events (left module), or just invoke its service commands (right module).

Listing 6.3 shows an example server that implements the MembershipListener interface, and exploits the MembershipService interface to become member of group 5.

Establishing the connections between the server and its associated set of protocol modules is done through the GroupManager object. The GroupManager object wraps the protocol modules and acts as an interface between the modules and the server. Initially, when the server requests group communication support it will invoke the getGroupManager() factory method, passing its own reference (**this**) (see Line 7 in Listing 6.3). Given the server reference, the GM is able to establish upcall connections between the server and the modules whose listener interfaces are implemented by the server.

On the other hand, establishing connections between the server and the service interfaces of modules are done on-demand by the server implementation itself. This is accomplished by using the GroupManager.getService() method shown in Figure 6.5 and in Lines 9-10 in Listing 6.3. Given a reference to the service interface of some module, the server can easily invoke service commands, e.g. the join() method in Line 12 in Listing 6.3.

Note that even though a server is associated with a set of protocol modules, it does not have to interact with any of the modules in the set. Mostly, the server will interact directly with only a small subset of the modules. Also, a particular server may be interested only in a small part of the methods declared in the listener interfaces, in which case it can simply provide an empty implementation for those methods, e.g. Lines 21-22 in Listing 6.3.

Listing 6.3: Example server using the MembershipService and MembershipListener.

```
1  public class ExampleServer
2    implements MembershipListener
3  {
4    public ExampleServer()
5    {
6      /* Obtain a group manager for this server object */
7      GroupManager gm = GroupManager.getGroupManager(this);
8      /* Obtain a reference for the membership service */
9      MembershipService pgms = (MembershipService)
10       gm.getService(MembershipService.class);
11     /* Join group 5 */
12     pgms.join(5);
13   }
14
15   /* Methods from the MembershipListener interface */
16   public void viewChange(View view)
17   {
18     System.out.println("New␣view␣installed:␣" + view);
19   }
20
21   public void prepareChange() {}
22   public void hasLeft() {}
23 }
```

6.4.4 External Entity to Module Interactions

Protocol modules may also interact directly with (possibly replicated) external entities. For instance, a protocol module could invoke methods on an external entity or vice versa. This interaction style is new and is useful for a number of purposes, such as event logging, event notifications or triggering some action, e.g. recovery or upgrade.

Note that we distinguish between remote peer module interactions as discussed in Section 6.4.2, and interactions with external entities. This is because peer module interactions assume the modules are in the same group, and hence view synchrony can be guaranteed, whereas interactions with external entities cannot provide such guarantees. Nonetheless, interaction with external entities has proven to be a vital interaction style for many of the subsystems discussed in later chapters.

External entities and modules can interact in both directions, as shown in Figure 6.6. Interaction with external entities relies on the dependable registry for looking up the

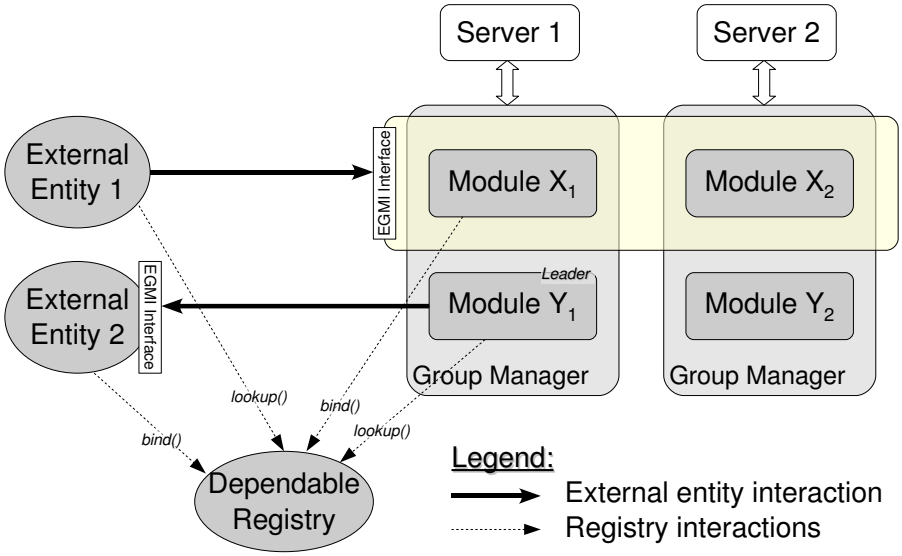


Figure 6.6: External entity to module interactions.

reference of the external entity (or the group of modules) with which to communicate. Prior to such lookups, the receiving end must bind() its reference in the dependable registry. The two interactions shown in Figure 6.6 are both based on EGMI, and hence the receiving end must include the ExternalGMIModule in its set of protocol modules.

External Entity 1 can invoke the EGMI interface of Module X to perform some operation implemented by the module. For example, the upgrade manager (the external entity) can multicast an upgradeRequest() to the UpgradeModule (Module X) associated with each of the server replicas, to request that the replica be upgraded to a new version. Further details about the upgrade approach is given in Chapter 12.

External Entity 2 implements a remote EGMI interface, which indicates that it is replicated using Jgroup. This allows a module to invoke methods on the external entity to perform some operation. For example, the ARM replication manager (the external entity) use this interaction style for receiving viewChange() events generated by the SupervisionModule (Module Y). Note that when using this interaction style, it is common that only the *leader* replica perform invocations so as to reduce the chance of performing duplicate invocations on the external entity. See Chapter 11 for additional details.

Note that although the above description has assumed the use of EGMI for communication with external entities, using standard RMI is also possible. In particular, the client-side group proxy (the external entity), discussed briefly in Section 3.3.2, use RMI to communicate with the ExternalGMIModule of one of the servers in the group. The ExternalGMIModule is responsible for performing invocations on the servers and returning the result to the client-side proxy.

6.5 The Dynamic Construction of Protocol Modules

The group manager encapsulates the set of protocol modules associated with an application. Previously however, the group manager provided only a fixed set of statically constructed protocol modules (services), i.e. those described in Chapter 3. This section describes the enhancements made to the group manager construction mechanism aimed at improving the flexibility in protocol module configuration. Protocol modules are now configured using the application-specific replication policy as discussed in Chapter 10. The policy supports specifying the set of protocol modules to be constructed, as well as supplying configuration parameters to the modules, e.g. timeout values.

Protocol modules are constructed dynamically at runtime based on the replication policy of the application requesting the construction of a GM. Line 7 in Listing 6.3 illustrates how the server interacts with the GM to perform the construction. This is essentially all a server developer needs to know about the construction of protocol modules. However, a developer of generic modules needs to have more intimate knowledge of the architecture.

To a module developer, the dynamic construction facility simplifies these tasks:

- Automatic construction of protocol modules.
- Establishing links between dependent modules.
- Establishing links between the server and its dependent modules.
- Reconfiguration of links for module substitution.

To take advantage of the dynamic construction facility, the module developer must adhere to the rules listed below:

1. The module must contain a single constructor, whose signature contains the set of services *required* by the module.

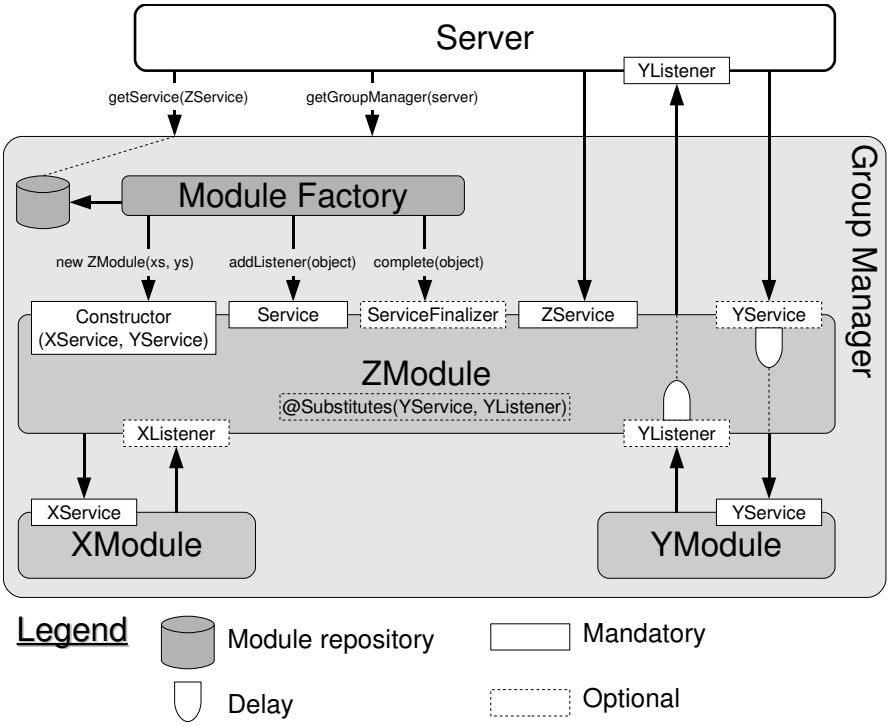


Figure 6.7: The module factory and interfaces used for module construction.

2. The module *must* implement the Service interface.
3. The module *may* implement the ServiceFinalizer interface.
4. The module *may* implement listener interfaces of other modules.
5. The module *may* declare that it *substitutes* the services/listeners provided by another module.

Figure 6.7 illustrates these rules in terms of interfaces. Solid boxes indicate required interfaces, while dashed boxes denote optional interfaces which may be implemented by a module depending on its requirements.

6.5.1 Module Instantiation

As shown in Figure 6.7, ZModule requires two other services, XService and YService, which are implemented by XModule and YModule, respectively. These two modules

must have been constructed prior to ZModule, and are passed as parameters to the ZModule constructor. The *module factory* uses Java reflection [6, Ch.16] to examine the constructor signature of the ZModule to determine its required module dependencies, and queries the *module repository* to obtain the required modules. If a required module is not found in the repository it will be created on-demand and stored in the repository. Note that cyclic module dependencies are not possible with this approach, i.e. if ZModule depends on XModule and vice versa, they cannot be constructed using the scheme above. However, it is still possible to manually implement dependency cycles through minor supplements to the mutually dependent modules.

Construction order may sometimes be important for correct functioning of a protocol stack. Construction follows the bottom-up order specified in the replication policy (see Chapter 10). Referring to Figure 6.1, this means that the DispatcherModule is constructed first, followed by the MembershipModule and so on. Note that the DispatcherModule does not depend on other GM modules, but is instead responsible for constructing or establishing connection with a daemon.

6.5.2 Link Configuration

Once all the protocol modules associated with an application have been instantiated, links between the modules are established by the module factory through the mandatory Service interface implemented by all modules. The addListener() method shown in Figure 6.7 serves two primary purposes:

- To establish upcall links with other modules and the server; links are only established with modules (or the server) implementing the listener interface associated with the module.
- To perform bootstrap operations that cannot be performed during module construction.

In Figure 6.7 the object passed to the addListener() method may be either the server object or a module. Note that the server object is always passed to the addListener() method, independent of it implementing the listener interface associated with the module. Thus the module can exploit the server reference type as a means to obtain necessary configuration data from the replication policy, e.g. timeout values, to configure/bootstrap the module. If the server object does not implement the listener interface of the module, it cannot receive any events from the module. Furthermore, the addListener() method may be invoked several times for distinct modules, allowing

multiple modules to receive the same set of events. The order in which addListener() is invoked follows the construction order defined above, with the server object passed in last.

6.5.3 Bootstrapping

Some modules may need to perform supplementary bootstrap operations after all the links have been established. The final task performed by the module factory is to find modules that implement the optional ServiceFinalizer interface, and invoke its complete() method to perform the final bootstrap operations. For instance, the server could configure its replication policy to automatically join() its group during the bootstrap phase, obsoleting Lines 8-12 in Listing 6.3. Joining the group requires that all the links have been set up between all the protocol modules, and hence it cannot be bootstrapped through the Service interface. Given this bootstrap mechanism, some modules may even be able to replace its service interface with an empty marker interface, and instead perform its operations automatically.

6.5.4 Event Interception

As advocated initially in this chapter, some modules need to intercept commands/events originated in other modules. Such interception may be necessary for a number of reasons, e.g. if delivery of events must be delayed until after the intercepting module has completed its tasks. For example, a total ordering module needs to delay the delivery of messages pending agreement among group members on the sequence in which to deliver messages.

Modules that wish to intercept the commands/events of another module must declare that it substitutes the other module. The @Substitute declaration uses Java annotations [6, Ch.15] to indicate which service and listener interfaces to substitute. As shown in Figure 6.7, the ZModule substitutes both interfaces associated with the YModule. The module factory will analyze the substitute declarations and reconfigure the links accordingly, hiding the presence of the YModule from other modules and the server.

Implementing a module which substitutes another can be accomplished by inheriting from the substituted module, or by wrapping it.

Note that it is essential that substituting modules be ordered appropriately in the replication policy so as to ensure correct interception.

6.5.5 An Example Protocol Module

Listing 6.4 shows the MembershipModule as an example of the core parts required to be implemented by a protocol module. Particular emphasis has been put on the handling of MembershipListeners. The module factory will discover modules (including the server) that implement the MembershipListener interface, and invoke the MembershipModule.addListener() method for each such module. In response to events received through the DispatcherListener implemented by the MembershipModule, the various modules registered for membership events are invoked through their corresponding viewChange() methods.

Otherwise, the MembershipModule implements the MembershipService interface, and through it also the Service and ServiceFinalizer interfaces (see Line 2 in Listing 6.1). Examining the constructor, notice also that the MembershipModule *requires* the services of DispatcherService. Although not shown, the MembershipModule will use the methods of the DispatcherService to send events to the daemon.

6.5.6 Impact of Dynamic Module Construction

The construction of protocol modules is a one-time operation performed when a replica is started. It will thus impact the recovery performance that can be obtained. However, the cost of constructing protocol modules dynamically vs. using a statically compiled set of modules is relatively small compared to the gain obtained. The average construction overhead is approximately 7 ms for the full set of Jgroup protocols (those described in Chapter 3). This overhead is insignificant compared to other delays contributing to the overall recovery performance, as discussed in Part IV.

Listing 6.4: Excerpt from MembershipModule.

```
public class MembershipModule
  implements MembershipService, DispatcherListener
{
  [...]
  private List<MembershipListener> membershipListeners =
    new ArrayList<MembershipListener>();

  public MembershipModule(DispatcherService dispatcher) { }

  // From Service interface; inherited from MembershipService
  public void addListener(Object listener)
  {
    if (listener instanceof MembershipListener
        && !membershipListeners.contains(listener))
      membershipListeners.add((MembershipListener)listener);
  }

  // From ServiceFinalizer; inherited from MembershipService
  public void complete(Object server) { }

  // From MembershipService
  public void join(int gid) { }

  // From DispatcherListener
  public void notify(Event event)
  {
    [...]
      case INSTALL_EVENT:
        handleInstallEvent((InstallEvent) event);
  }

  private void handleInstallEvent(final InstallEvent event)
  {
    final View view = event.getView();
    for (MembershipListener listener : membershipListeners)
      listener.viewChange(view);
  }

} // END MembershipModule
```

Chapter 7

Adaptive Protocol Selection

Middleware frameworks for building dependable distributed applications often provide a collection of *replication protocols* supporting varying degrees of consistency. Typically, providing strong consistency requires costly[1] replication protocols, while weaker consistency often can be achieved with less costly protocols. Hence, there is a tradeoff between cost and consistency involved in the decision of which replication protocol to use for a particular server. But, perhaps more important is the behavioral aspects of the server. For instance, the server may be intrinsically non-deterministic in its behavior, which consequently rules out several replication protocols from consideration, e.g. atomic multicast.

The topic of this chapter is the architecture for selection of replication protocols. The architecture is built on the external group method invocation approach discussed in Chapter 3. The revised EGMI architecture makes it very easy to add new replication protocols to the system, with no changes to the core toolkit. Protocol implementations are picked up automatically. Furthermore, the architecture also boosts the flexibility in using the various protocols by allowing *each method* to declare its own replication protocol. Support for two new replication protocols have also been added to the new EGMI architecture; *atomic multicast* and *leadercast*. The latter is a variant of passive replication and permits servers with non-deterministic behavior, whereas atomic multicast can be viewed as a kind of active replication, and hence does not tolerate servers being non-deterministic. The new architecture can also easily be enhanced to support adaptive protocol selection based on runtime changes in the environment.

[1] Costly in terms of communication overhead.

This chapter is organized as follows: Section 7.1 motivates the need for a revised EGMI architecture and provides guidelines for choosing appropriate replication protocols for fault tolerant servers. In Section 7.2 the EGMI architecture is presented, while in Section 7.3 the protocol selection mechanism is covered. The leadercast replication protocol is covered in Section 7.4, and Section 7.5 covers the atomic replication protocol. Finally, Section 7.6 discusses potential enhancements to the architecture that would enable support for adaptive runtime selection of protocols.

7.1 Motivation

The principal motivation for redesigning the EGMI architecture is to improve the flexibility in choice of replication protocols, so as to reduce the resource consumption of dependable applications as much as possible.

In many fault-tolerant systems, different replication protocols are supported at the *object* level [93, 104], meaning that all the methods of a particular object must use the same replication protocol. Jgroup takes a different approach: when implementing an external interface, the invocation semantic of each individual method can be specified separately using Java annotations. This allows for greater flexibility as various methods may need different semantics. Hence, developers may select the appropriate invocation semantics at the method level, and even provide different implementations with alternative semantics.

By exploiting knowledge about the semantics of distributed objects, the choice of which replication protocols to use for the various methods can be used to obtain a performance gain over the traditional object level approach. Similar ideas were proposed by Garcia-Molina [46] to exploit semantic knowledge of the application to allow nonserializable schedules that preserve consistency to be executed in parallel as a means to improve the performance for distributed database systems. OGS [41, 42] also allows each method of a server to be associated with different replication protocols, but this must be explicitly encoded for each method through an intricate initialization step. The Jgroup approach presented herein is much easier to use as it exploits the Java annotation feature to mark methods with the desired replication protocol. The Spread [3] message-based group communication system can also be used to exploit semantic knowledge, since each message can be assigned a different replication protocol. JavaGroups [18] on the other hand would have required separate channels for each replication protocol. Unlike Jgroup however, neither of these two systems are aimed at RMI based systems.

A common example in which application semantic knowledge can be exploited is a replicated database with read and write methods. Often a simple *Read-One, Write-All* (ROWA) replication protocol [125] can then be used and still preserve consistency. A ROWA replication protocol can easily be implemented using anycast for read methods and either multicast, atomic, or leadercast for write methods. On the other hand, replication protocols which operate at the object level require that also simple read-only methods use the *strongest* replication protocol required by the object to preserve consistency.

Felber et al. [44] discuss various semantic properties that can be used to reduce the resource consumption for certain operations (methods) on a replicated object. These properties are summarized here for convenience:

1. **Read-only methods** do not modify the shared state of a replicated object. Using costly replication protocols for read-only methods are rarely necessary, but may sometimes be required if there is a causal relationship between read-only and write invocations. In Jgroup read-only methods should use the *anycast* protocol.
2. **Idempotent methods** can be invoked twice with the same arguments and still have the same effect as calling it only once. This property is useful since it permits clients to reissue a request without harmful effects.
3. **Deterministic methods** are such that their effects on the shared state and output to clients depend only on the initial state of the object, and the sequence of methods performed on the object. For active replication, deterministic behavior is mandatory. Jgroup provides the *atomic* protocol for deterministic methods that modify the shared state and require strong consistency, while the *leadercast* protocol can be used for non-deterministic methods.
4. **Commutative methods** A pair of methods are commutative if it does not matter in which order they are called on the object. Jgroup provides the *multicast* protocol for commutative methods that modify the shared state, but require only weak consistency.
5. **Parallelizable methods** A pair of methods are said to be parallelizable if their concurrent execution is equivalent to a serial execution of both methods. For instance, methods that only modify disjoint parts of the shared state of a replicated object are often parallelizable.

Table 7.1: The properties associated with the various replication protocols.

	Anycast	**Multicast**	**Leadercast**	**Atomic**
Read-only	Suitable	Unsuitable	Unsuitable	Unsuitable
Write	Unsuitable	Suitable	Suitable	Suitable
Commutative	Suitable	Suitable	—	—
Consistency	Weakest	Weak	Strong	Strong
Comm. overhead	Low	Medium	Medium	High
Comp. resources	Low	High	Medium	High
Failover delay	Medium	Medium	High	High
Determinism	Not Required	Required	Not Required	Required

Note that care should be taken when developing applications that combine the various replication protocols provided by Jgroup. For example, if two methods modify intersecting parts of the shared state, both methods should use the same replication protocol.

Table 7.1 summarizes the properties of the various replication protocols supported by Jgroup/ARM. The table along with the semantic properties discussed above is meant to serve as a guideline for application developers when deciding which replication protocol is appropriate for each method of their servers.

A few things to note: Commutative write operations may also use leadercast or atomic, but it may not be necessary. Moreover, the failover delay is actually quite high for all the supported protocols. This is because the client is considered external to the group and as such may take longer to detect a server failure. Additional details are provided in the following sections.

7.2 The EGMI Architecture

The external group method invocation (EGMI) architecture has been redesigned to better cope with application-specific requirements, such as adaptive selection of replication protocols on a per invocation basis. There are three principal reasons for the redesign, as the original EGMI implementation suffered from a number of problems as listed below:

1. Only two fixed protocols were supported; anycast and multicast. Adding new replication protocols was quite cumbersome, and the choice of protocol was fixed at service design time.

2. Improper use of the exception declaration mechanism; protocol annotation was implemented by augmenting the throws clause of methods with a protocol specific exception (see Section 3.3.2.2).
3. There was no updating of the client-side group membership information. In fact, it was impossible to fully support client-side view updating (see Chapter 9) with the previous EGMI design.

Note that adding alternative replication protocols through the protocol module mechanism discussed in Chapter 6 could also be done, e.g. by substituting the MulticastModule with a TotalOrderModule. However, that would have caused all multicast methods to use the TotalOrderModule, even if only some of the methods needed to maintain strong consistency.

The redesigned EGMI architecture aims to provide:

1. Improved flexibility and efficiency by means of a customized RMI layer.
2. Flexibility to add new replication protocols.
3. Runtime adaptive selection of replication protocol.
4. Improved client-side view updating.

The second topic is covered in the next section, while the latter is covered in Chapter 9. In this section, the overall EGMI architecture is presented along with the details of the customized RMI layer. Runtime adaptive protocol selection is covered in Section 7.6.

Figure 7.1 illustrates the EGMI architecture focusing on the high-level interactions between the various modules and external entities in the system. The figure illustrates interactions involved in a multicast invocation; the anycast invocation interactions are much simpler as it does not need to multicast the invocation.

Clients communicate with an object group through a *two-step* approach, except for the anycast semantic. Currently, two communication steps are required for interactions that involve all group members through the MulticastModule. Direct client multicast could be implemented [118], however not without adding a substantial amount of code to the client-side runtime. In addition, "distant" clients may not be able to exploit IP multicast, due to its limited deployment in Internet routers.

Returning to Figure 7.1; the ExternalGMIModule acts as the *server-side proxy* (representative) for clients communicating with the object group. The server representing

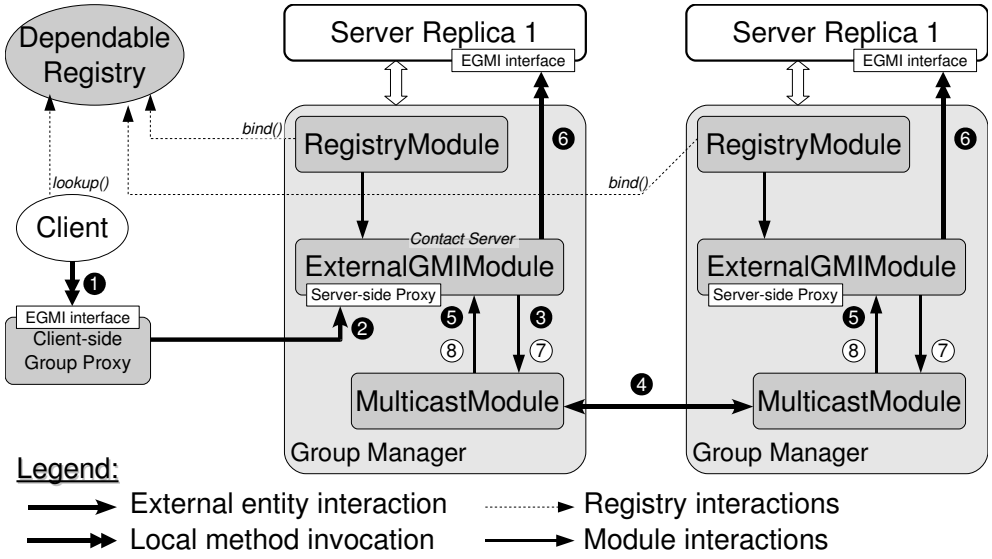


Figure 7.1: The external GMI architecture.

the group is called the *contact server*. The choice of contact server is made (on a per invocation basis) by the *client-side proxy*, and different strategies can easily be implemented depending on the requirements of the replication protocol being used. The general strategy used by both anycast and multicast is to choose the contact server arbitrarily, while leadercast always selects the group leader. However, in the presence of failures an arbitrary server in the group is selected.

As shown in Figure 7.1, before a client can invoke the object group, each member of the group must bind() its reference in the dependable registry. The RegistryModule is responsible for constructing its local part of the client-side proxy based on input from the ExternalGMIModule, and to pass it on to the registry. This is handled automatically in the module bootstrap phase. The client can then perform a lookup() to obtain the client-side proxy encompassing all group members.

The client-side proxy provides the same EGMI interface as the server. Hence, given the client-side proxy object, the client can invoke local methods on it (❶). The proxy will encode such invocations into remote communications (❷), and ultimately complete the invocation by returning a result to the client. The ExternalGMIModule exploits the MulticastModule to send multicast messages (❸,❹,❺) to all group members. This is followed by the invocation of the encoded method (❻) on all members, and returning the results back to the contact server (⑦,⑧). The contact server is responsible

for returning a selected result back to the client. Using the MulticastModule allows enforcement of view synchrony for EGMI invocations.

7.2.1 The Client-side and Server-side Proxies

In this section, we discuss the internal details of the client-side and the server-side proxies, or more precisely the internals of the ExternalGMIModule. The internals are basically a customized version of the Jini Extensible Remote Invocation (JERI) protocol stack discussed in Section 2.2.3.1. By using JERI, several cumbersome and inflexible extensions to the plain Java RMI model used in the original Jgroup EGMI implementation were eliminated. The JERI protocol stack shown in Figure 2.6 has been extended with functionality to support group communication. The new protocol stack is illustrated in Figure 7.2, and shows that all layers have been retrofitted with group communication support, except for the transport layer. Currently, a TCP transport layer is used. It should be noted that this is only for the transport between clients and the contact server, whereas the contact server uses the MulticastModule if multicast communication is required.

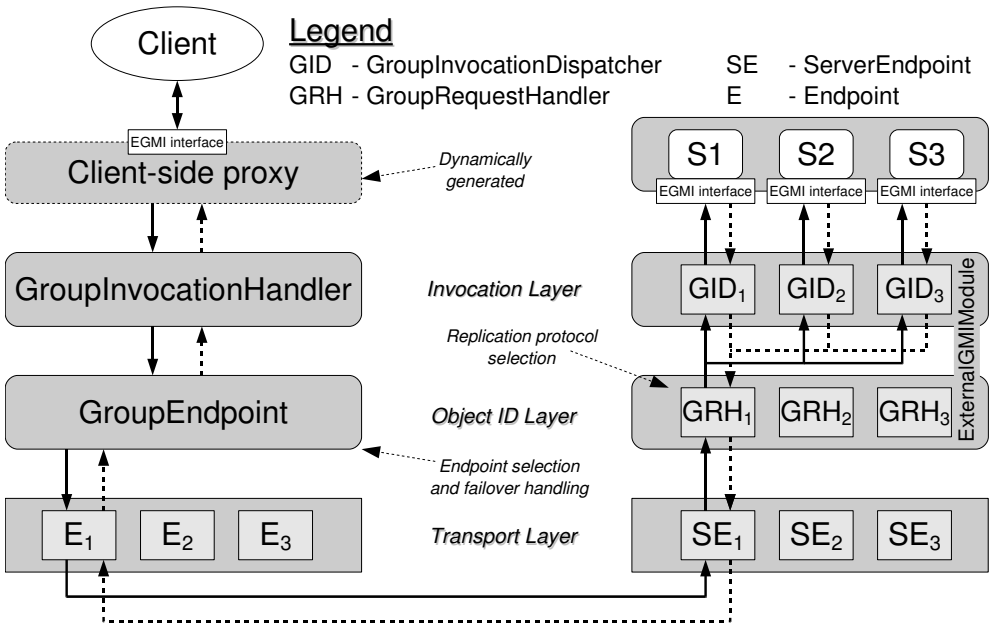


Figure 7.2: The EGMI protocol stack.

The GroupInvocationHandler shown in Figure 7.2 is mainly responsible for marshalling and unmarshalling invocations. When invoked by the client-side proxy, internal tables are queried to determine the invocation semantic of the method being invoked. Knowing the invocation semantic on the client-side is useful so that efficient marshalling can be performed, e.g. multicast invocations need not be fully unmarshalled at the server-side until received by the GroupInvocationDispatcher.

The GroupEndpoint is used to represent the current group membership, and stores a single Endpoint for each member of the group. Each Endpoint object represents the transport between the client and the corresponding ServerEndpoint. The GroupEndpoint is also in charge of selecting which of the endpoints to use for a particular invocation, based on the semantic declared for the method being invoked. Furthermore, the GroupEndpoint must keep its membership list up-to-date with respect to the current server-side membership. The latter issue is covered in Chapter 9.

When the GroupRequestHandler (GRH) receives an invocation, the invocation semantic is extracted from the data stream. Depending on the invocation semantic, the invocation is passed on to a protocol-specific invocation dispatcher (protocol dispatchers are discussed in the next section). For the purpose of this discussion, the protocol dispatcher is assumed to be multicast (corresponding to that of Figure 7.2). Hence, the stream is passed on to the MulticastModule, and finally to the GroupInvocationDispatcher (GID) which takes care of the unmarshalling and invocation of the method on the remote server objects.

As Figure 7.2 shows, the results are returned to the contact server, which finally returns the result(s) to the client.

7.3 Replication Protocol Selection

As discussed in the beginning of this chapter, each method invocation can use a distinct invocation semantic. The choice of invocation semantic is usually made by the server developer at design time. This is done by prefixing each method with a annotation marker indicating the replication protocol to use for each particular method. The RegistryImpl server shown in Listing 7.1 illustrates how the developer can indicate the replication protocol to use for the various methods.

Note that it is also possible to declare protocol annotations in the interface, e.g. the DependableRegistry interface. However, annotation markers declared in the server implementation takes precedence over those that may be declared in the interface.

Listing 7.1: Skeleton listing of the DependableRegistry implementation.

```
public final class RegistryImpl
  implements DependableRegistry
{

  // Methods from DependableRegistry
  @Multicast public IID bind(String name, Entry entry)
   throws RemoteException, AccessException

  @Multicast public void unbind(IID iid)
   throws RemoteException, NotBoundException, AccessException

  @Anycast public String[] list()
   throws RemoteException

  @Anycast public Remote lookup(String serviceName)
   throws RemoteException, NotBoundException

} // END RegistryImpl
```

This makes it easy to provide alternative implementations of the same interface with different invocation semantics for the various methods declared in the interface. For example, if a particular implementation wants to provide stronger consistency for some methods.

Figure 7.3 depicts the ExternalGMIModule and its protocol selection mechanism. Each protocol must implement the ProtocolDispatcher interface through which invocations are passed before they are unmarshalled. This allows the protocol to multicast the unmarshalled invocation to the other group members before unmarshalling is done in the GroupInvocationDispatcher. However, the stream received by the GroupRequestHandler is partially unmarshalled to obtain information necessary to *route* the message to the appropriate protocol dispatcher instance.

The *protocol repository* shown in Figure 7.3 holds a mapping between the annotation marker (a method's invocation semantic) and the actual protocol instance. The repository is queried for each invocation of a method.

7.3.1 Supporting a New Protocol

To support new EGMI replication protocols, two additions are required:

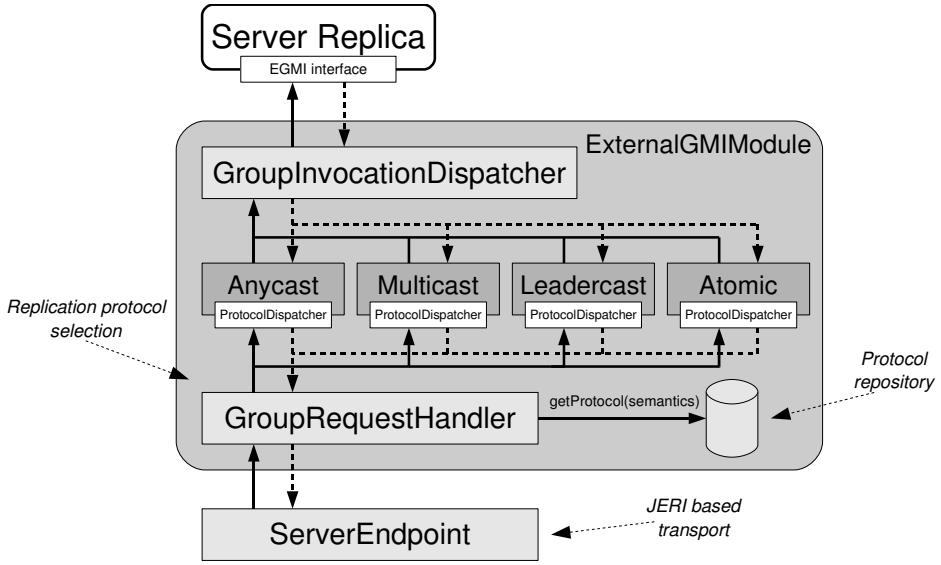


Figure 7.3: EGMI replication protocol selection.

1. A new annotation marker must be added, allowing servers to specify the new protocol.
2. The actual protocol implementation.

Listing 7.2 shows the annotation marker for the @Atomic replication protocol. To support runtime protocol selection, the retention policy of the marker must be set to RUNTIME to allow reflective access to the marker. Furthermore, the target element type is set so that the marker only applies to METHOD element types. For additional details about the Java annotation mechanism see [6, Ch.15].

Listing 7.2: The @Atomic annotation marker.

```
package jgroup.core.protocols;

@Documented
@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Atomic { }
```

A new protocol implementation must implement the ProtocolDispatcher interface (see Listing 7.3). The package pickup location for protocols is easily configured through a Java system property; currently only a default location is used. Hence, adding a new protocol is done by placing the protocol in the pickup location. The ExternalGMIModule takes care of constructing the protocol.

Listing 7.3: The ProtocolDispatcher interface.

```
package jgroup.relacs.gmi.protocols;

public interface ProtocolDispatcher
{
  public InvocationResult dispatch(InputStream in)
    throws IOException;

  public void addListener(Object listener);
}
```

Constructing an EGMI replication protocol is similar to the construction of GM protocol modules, discussed in Chapter 6. A replication protocol may specify the set of modules that it requires in its constructor signature, and it may also implement the listener interfaces of modules. In addition, the constructor will typically also include the GroupInvocationDispatcher, through which the protocol can perform invocations on the local server.

The ExternalGMIModule will reflectively [6, Ch.16] analyze the server implementation (or its EGMI interfaces) to determine the invocation semantics of its methods. Methods whose invocation semantic is unspecified defaults to @Anycast. This analysis is done during the ExternalGMIModule bootstrap phase, and the information is kept in internal tables for fast access during invocations. This table is also kept in the client-side proxy to determine the invocation semantic of the method being invoked. Replication protocols are constructed on-demand when analyzing the methods of the server. Hence, only the required protocols are constructed. Note that GM protocol modules (not just the server) may implement EGMI interfaces that are picked up by the ExternalGMIModule during the bootstrap phase.

7.3.2 Concurrency Issues

Note that a protocol instance may be invoked concurrently by multiple clients, and care should be taken when developing a replication protocol to ensure that access to

protocol state is synchronized. Furthermore, the EGMI architecture is designed for multithreading, and hence it does not block concurrent invocations using the same or different protocols. It is the responsibility of the server developer to ensure that access to server state is synchronized. However, invocations received while a new view is pending are blocked temporarily and delivered in the next view. This is necessary to avoid that invocations modify the server state while the state merge service (see Section 3.3.3) is active. Optimizations could be implemented to avoid blocking for protocols such as anycast aimed at read-only methods.

7.4 The Leadercast Protocol

The leadercast protocol presented in this section is a variant of the passive replication protocol [26, 51] discussed in Section 2.4.2. The principal motivation to provide this protocol is the need for a strong consistency protocol that is able to tolerate non-deterministic operations. The main difference between leadercast and the passive replication protocols described in literature [26, 51] is optimizations in scenarios where the leader has crashed. That is how to convey information about the new leader to clients, and how to handle failover. These optimizations are possible due to the client-side view updating technique discussed in Chapter 9.

Figure 7.4 illustrates the leadercast protocol, when the client knows which of the group members is the leader. In this case, the protocol is as follows:

1. The client sends its request to the group leader (over a unicast channel).
2. The leader process the request, updating its state.
3. The leader then multicasts an update message containing ⟨Result, StateUpdate⟩ to the followers (backups).
4. The followers modify their state upon the reception of an update message, and reply with an acknowledgement to the leader.
5. Only when the leader have received an acknowledgement from all live follower replicas, will it return the Result to the client.

Result is the result of the processing performed by the leader, while StateUpdate is the state (or a partial state) of the leader replica after the processing. A partial state may for instance be the portions of the state that have been modified by the leadercast

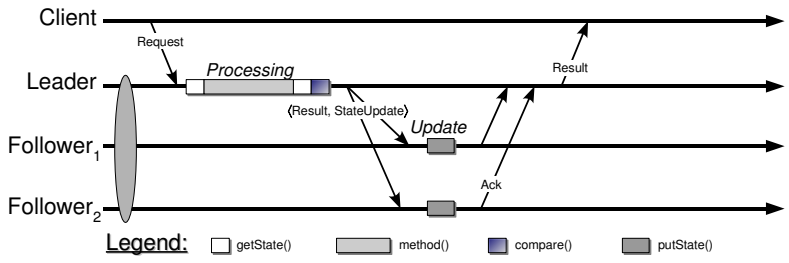


Figure 7.4: The Leadercast protocol with leader receiver.

methods. Providing these state update routines implies application involvement by implementing a StateUpdateListener interface.An alternative approach could be to use reflective analysis of the server state to determine and only transport the difference between the old and new state to the followers.

Notice the compare() method performed at the end of the processing. This is used to compare the state of the server before and after the invocation of method(), and if the state did not change due to the invocation, there is no need to send the update message, as shown in Figure 7.5.

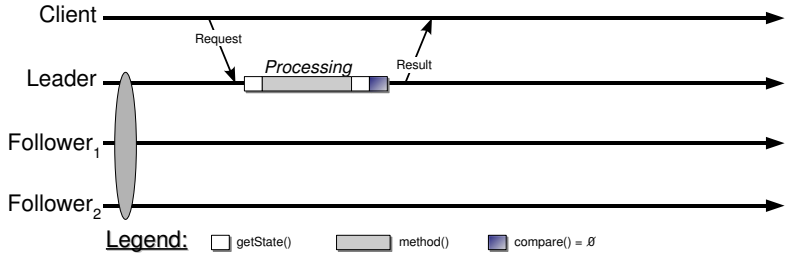


Figure 7.5: The Leadercast protocol with no state change.

Note that the Result part of the update message is necessary in case a follower is promoted to group leader, and needs to emit the Result to the client in response to a reinvocation of the same method. This can only happen if the leader fails, causing the client to perform a failover by reinvoking the method on another group member, as shown in Figure 7.6. Hence, the followers needs to keep track of the result of the previous invocation made by clients. A result value can be discarded when a new invocation from the same client is made, or after some reasonable time longer than

the period needed by the client to reinvoke the method.

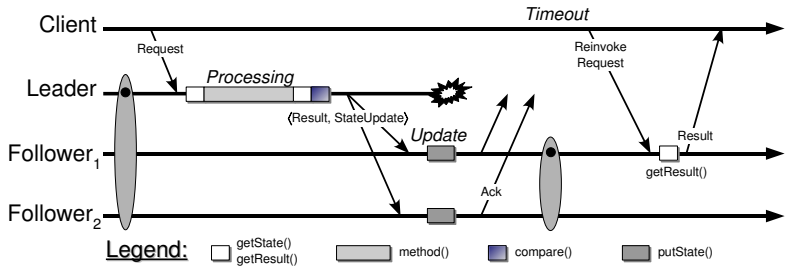


Figure 7.6: The Leadercast protocol with failover.

As depicted in Figure 7.6, the failure of the leader causes the membership service to install a new view. Client invocations may be received before the new view is installed, however, they will be delayed until after the view has been installed, as discussed in Section 7.3.2. The follower receiving the reinvocation of a previously invoked method will simply return the result to the client along with information about the new group leader.

If the follower receiving a reinvocation of a previously invoked method is not the new leader, the invocation will be forwarded to the current leader, as shown in Figure 7.7. This can happen if the leader failed before the followers could be informed about the original invocation. Note that this forwarding to the current leader will only occur once per client, since the result message contains information about which member is the new leader, and hence the client-side proxy can update its contact server.

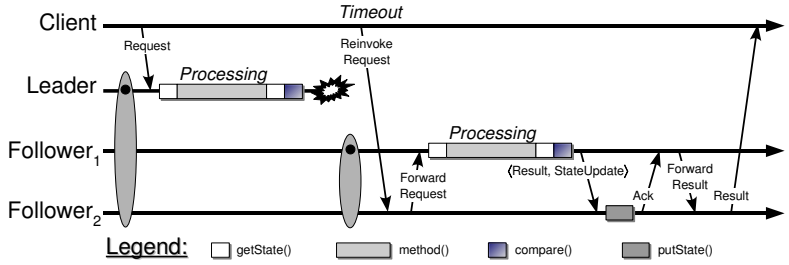


Figure 7.7: The Leadercast protocol with follower receiver after a failover.

As discussed above, the client-side group proxy is responsible for selecting the contact server. For the leadercast protocol, the group leader (primary) is selected unless

it has failed. The contact server selection strategy is embedded in the invocation semantic representation associated with each method. When the client detects that the group leader has failed, the choice of contact server is random for the first invocation; the new leader is then obtained from the result value of the invocation and future invocations are directed to the current leader.

7.5 The Atomic Multicast Protocol

The atomic multicast protocol implemented in the context of this thesis is based on the ISIS total ordering protocol [23]. A description of the algorithm can also be found in [32], hence only a brief description is provided herein. The protocol is useful to ensure that methods that modify the shared server state do so in a consistent manner. Methods using the atomic protocol must behave deterministically to ensure consistent behavior.

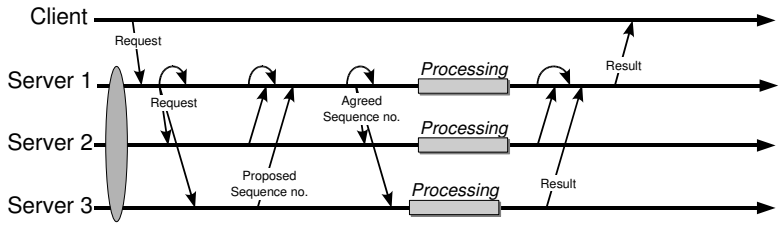


Figure 7.8: The Atomic multicast protocol.

The protocol is a *distributed agreement protocol*[2] in which the group members collectively agree on the sequence in which to perform the invocations that are to be ordered. Figure 7.8 shows the protocol. In the first step, the client sends the request to a contact server (the choice of contact server is discussed below). The contact server forwards the request to the group members, each of which respond with a *proposed sequence number*. The contact server then selects the *agreed sequence number* from those proposed and notifies the group members; the highest proposed sequence number is always selected. Finally, when receiving the agreed sequence number each member can perform the invocation and return the result(s) to the contact server, which will relay it to the client.

[2] In [36] it is called *destination agreement*.

The contact server selection strategy is random for load balancing and fault tolerance purposes. The contact server acts as the entity that defines the ordering of messages, and serves this function for all invocations originated by clients using it as the contact server. Since the choice of contact server is random, the same client may choose a different one for each invocation that it performs. It follows that also different clients will use different contact servers.

An alternative contact server selection strategy is to always select the same server (the leader) to do the message ordering. By doing so, a *fixed sequencer* protocol requiring less communication steps can be implemented. The fixed sequencer and other total ordering protocols are discussed in [36].

Figure 7.9 illustrates one scenario in which the contact server fails before completing the current ordering. The client detects the failure of the contact server, and sends the request to an alternative contact server. In this particular scenario, the remaining servers needs to rerun the agreement protocol. However, had the contact server failed after completing the agreement protocol, but before emitting the result to the client, the new contact server must emit the previous result in response to a reinvocation.

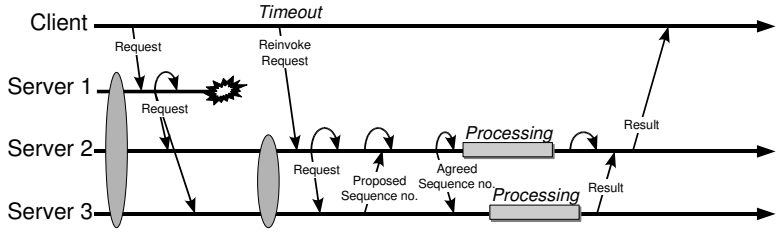


Figure 7.9: A failover scenario with the Atomic multicast protocol.

The two-step communication approach used for EGMI between the client and the group members precludes the provision of a true active replication scheme, as discussed in Section 2.4.1. In particular, the client-side proxy will not receive replies directly from all the servers, and thus cannot mask the failure of the contact server towards the client-side proxy. Hence, if the contact server fails during an invocation, the client-side proxy is required to randomly pick another contact server and perform a reinvocation. The failure of the contact server, however, is still masked from the actual client object. But the disadvantage is that the failover delay of the atomic approach is equivalent to that of the leadercast approach when the contact server fails. However, one way to provide true active replication is to let clients become (transient) members of the object group prior to invoking methods on it, allowing clients

to receive replies from all members and not just the contact server.

It is foreseen that the client-side proxy in the future can hide the fact that it has joined the object group, from the client object before performing an invocation, e.g. by annotating the method with @Atomic(join=true). An optional leaveAfter attribute could also be provided indicating the number of invocations to be perform before the client-side proxy requests to leave the group. This way true active replication can be provided also to clients.

7.6 Runtime Adaptive Protocol Selection

Another useful mechanism that can easily be implemented in this architecture is support for *dynamic runtime protocol selection*. Dynamically changing the replication protocol of methods at runtime is useful for systems that wish to dynamically adapt to changes in the environment or to handle runtime changes in the requirements. For instance, a server may decide to change its replication protocol for certain methods to improve its response time, if the system load increases. One might also imagine a special module that can configure the replication protocols of the server remotely from ARM or the management client to adapt to changing requirements. For example, if moving to more powerful hardware, one can simply migrate replicas to the new hardware, followed by a change of the replication protocol to use for certain methods.

This section briefly outlines how this feature can be implemented.

First, a @Dynamic marker is needed, which must be added to methods that should support dynamic reconfiguration. Next, the Dynamic replication protocol must be implemented, which is simply a wrapper for the other supported protocols. The Dynamic protocol must maintain a mapping for each @Dynamic method and its currently configured invocation semantic. By default, methods that declare @Dynamic should be configured with the @Anycast semantic, unless the marker is parameterized with the desired default protocol, e.g. @Dynamic(protocol=@Leadercast).

The Dynamic protocol can be exposed through the GM by providing a DynamicReplicationService interface through which methods can have their invocation semantics updated at runtime. This allows the server or other protocol modules to access the DynamicReplicationService, and thus implement update algorithms that can reconfigure the replication protocol of individual methods at runtime. One scheme could be to change the replication protocol of certain methods based on the size of the group. For

example, if the group only has three members or less then @Atomic is used; if it has more than three members then @Leadercast is used.

Another, more subtle use of this feature relates to a client designed for testing the performance of various replication protocols. The server can then simply implement a set of test methods, each declaring the @Dynamic marker, whereas the client can invoke a special method to set the appropriate replication protocol to be tested, before invoking the actual test methods on the server. To allow clients to reconfigure the replication protocol of methods, the server (or a module) must provide a remote interface (e.g. by exporting the DynamicReplicationService interface) through which clients can update the invocation semantics of the server-side methods.

Chapter 8

Enhanced Resource Sharing

As discussed in Section 1.2.1, this work assumes that nodes belong to a target environment, and each node is expected to be capable of hosting multiple services. Hence, the Jgroup model must support allocation of multiple replicas to the same node. However, in the original Jgroup prototype, placing two or more services on the same node required the allocation of distinct port numbers for each service, causing intricate port management issues. More importantly, only a single replica could connect to the same Jgroup daemon, i.e. there was a one-to-one mapping between the replica (group manager) and the daemon. This one-to-one mapping would lead to multiple instances of resource demanding components, in particular the inter-daemon failure detection component.

The aim of this chapter is to present the enhancements made to the Jgroup daemon architecture to improve its resource sharing properties, overcoming the above mentioned problems. In fact, to enable the ARM framework to place multiple application replicas of distinct types on the same node, this change was necessary. In addition, the chapter briefly covers the low-level communication infrastructure needed to support multicast communication in a wide area network (WAN) setting.

Chapter outline: Section 8.1 gives additional motivation and presents the new daemon architecture. Section 8.2 briefly explains the inter-daemon and group manager – daemon communication infrastructure. Three distinct allocations of group managers (replicas) to daemons in the target environment, each with different scalability and failure independence properties are discussed in Section 8.3. Section 8.4 presents the algorithm used to discover and create site-specific daemons according to some policy, and in Section 8.5 the two-way daemon – group manager failure detection scheme is presented.

8.1 The Jgroup Daemon Architecture

To overcome the limitations discussed initially, the Jgroup daemon architecture has been revised to take into account resource reuse by allowing multiple application replicas to share the same daemon through their local group manager. In doing so, the need to configure several port number allocations per node is also eliminated since multiple replicas on the same node can share the same daemon.

As mentioned in Chapter 3, the core group membership and reliable multicast communication facilities are implemented as part of the daemon, although application replicas only access these services through their respective group manager interfaces. In addition, the daemon also implements failure detection facilities that are not exposed directly through the group manager.

There are a number of compelling reasons for separating the daemon functionality from the group manager module architecture:

- The number of messages that needs to be exchanged is reduced, compared to the case when all group managers on the same node replicate failure detection, exchange of synchronization, reachability and routing information.
- Messages from local server replicas are multiplexed over a single connection, reducing the number of connections needed between each pair of daemons.
- Scalability to very large groups is easily accomplished by configuring the system to use only a few daemons at each site, instead of one daemon per node. This will reduce the cost of the view agreement protocol, and also the number of messages that needs to be exchanged. A similar approach to scalability of groups is used in Spread [3] and Moshe [65].

Figure 8.1 shows one possible arrangement of Jgroup daemons (JD) on a set of nodes within two sites. The figure also shows the interactions that occur between the various daemons and between the GMs and their associated daemons.

The target environment defines the set of sites and nodes that comprise the system. A *site* is defined as a collection of nodes that are tightly connected. Typically, a local area network (LAN) is used to connect the nodes in a site, and often a site corresponds to the DNS domain of the LAN associated with the site. Each site may support a configurable number of daemons, according to a system-specific scaling policy discussed in Chapter 10. However, at most one daemon per node is allowed in the same target environment.

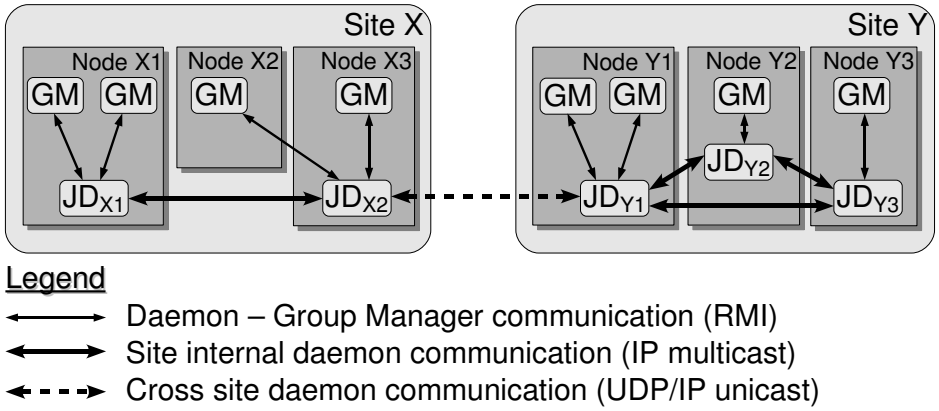


Figure 8.1: Overview of Jgroup daemon interactions.

In Figure 8.1 Site X has only two daemons, and GMs on nodes without a daemon, e.g. Node X2, have to connect to a remote daemon within the site, e.g. JD_{X2}. On the other hand, Site Y has configured its scaling policy to let each node have its own daemon. As Figure 8.1 also illustrates, multiple GMs may reside on the same node, e.g. Node X1, and connect to a single local daemon. In fact, GMs can connect to daemons anywhere within the site, but a local daemon will always be used if one exists. Each site must have at least one daemon.

8.2 Daemon Communication

Communication between a group manager and a daemon is based on reliable unicast communication (Java RMI). Inter-daemon communication use IP multicast between daemons within the same site (LAN), whereas communication between daemons residing at separate sites occur using a reliable unicast protocol implemented on top of UDP/IP.

8.2.1 Inter-Daemon Communication

The purpose of inter-daemon communication is primarily to provide reliable multicast capabilities within a group of daemons, and to keep track of node availability (or more precisely daemon availability). GMs connected to a daemon can send multicast messages within its group, and track the group membership.

As mentioned above, inter-daemon communication is based on a combination of IP multicast within a site and UDP/IP unicast when communicating across sites. Note that although not shown in Figure 8.1, all daemons in one site is able to communicate directly with all other daemons, including those at remote sites. Hence, all daemons keep track of all daemons at all sites. However, for a given round of communication with daemons in a remote site, the communication is mediated by a *contact daemon*, as illustrated in Algorithm 8.1. Hence, communication across sites requires two communication steps to multicast a message m to a group of daemons.

In Figure 8.1, the *sending daemon* is JD_{X2} and the contact daemon is JD_{Y1}.

The reasons for this two-step approach are twofold:

- WAN based IP multicast is beyond the control of our framework since multicast has limited deployment in Internet routers.
- Mediating traffic through a contact daemon allows additional resource sharing.

Algorithm 8.1 The two-step WAN multicast algorithm.

```
1:  Initialization:
2:    sites ← Config.getSites()                    {The set of sites}
3:    localSite ← Config.getLocalSite()            {The local site}
4:  void multicast(Message m)                      {MULTICAST AT SENDING DAEMON}
5:    foreach site ∈ sites
6:      if site.isLocalSite()
7:        site.multicast(m)
8:      else
9:        daemon ← site.selectDaemon()
10:       daemon.send(m)                           {Unicast send to contact daemon}
11: void receive(Message m)                        {RECEIVE AT CONTACT DAEMON}
12:   localSite.multicast(m)
```

The algorithm works as follows: for the local site, send m by way of reliable multicast; for all other sites, select a contact daemon to which m will be sent over a unicast channel. In the second step, the contact daemon simply multicasts m within its local site. For every message sent, the sending daemon selects a new contact daemon for each of the remote sites in the target environment. The choice of contact daemon is arbitrary, as a means of load balancing between the daemons at each site. However, only daemons with good reachability characteristics are selected. For instance, a daemon being suspected of having crashed will not be selected until its perceived reachability improves beyond some limit.

The reliable multicast communication layer in the daemon was implemented by Salvatore Cammarata [27] in the context of his masters thesis.

8.2.2 Group Manager – Daemon Communication

Group managers interact with their selected daemon by means of Java RMI, as shown in Figure 8.2. Events are passed back and forth between the daemon and the GMs using the dispatch() and notify() methods. In addition, the daemon can provide the number of members() currently connected to it, while the GM provides a ping() method that is used for failure detection (see Section 8.5).

The choice of Java RMI for this communication leg is mostly for flexibility and convenience in the prototype implementation, and may not be optimal with respect to performance when the daemon and its GMs are located in separate JVMs. However, in a WAN setting, the additional cost of RMI is marginal in comparison with the latencies involved in wide-area communication. Table 8.1 presents some simple measurements of the overhead due to RMI communication.

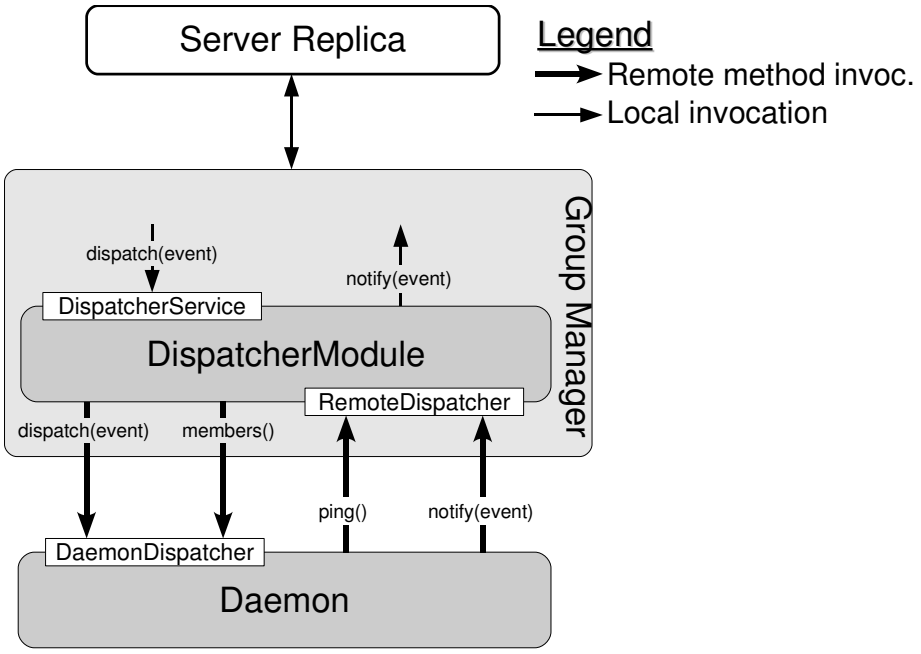


Figure 8.2: Group manager – daemon communication interfaces.

8.3 Daemon Allocation Schemes

Failure independence of co-located application replicas and the scalability in terms of number of group members have bearings on the choice of how group managers are allocated to daemons. Recall that there is a one-to-one mapping between application replicas and their associated GMs. As shown in Figure 8.1 there are several possible ways to allocate GMs (replicas) to daemons in the target environment.

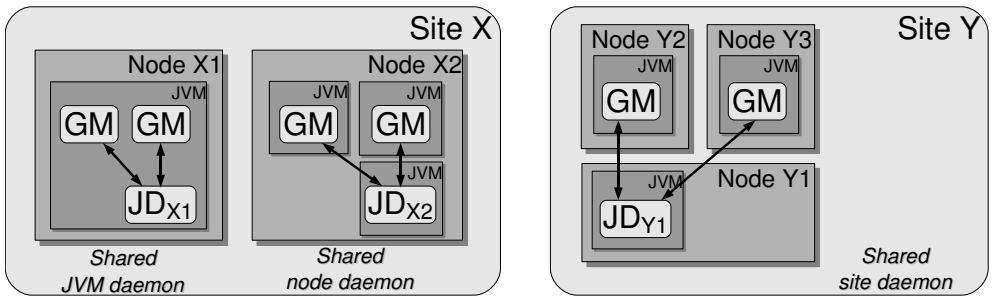


Figure 8.3: Group manager allocation mappings to daemon.

The following lists allocation mappings that are supported and briefly discuss their pros and cons.

1. **Shared JVM daemon** Group managers share the same JVM as the daemon, as shown on Node X1 in Figure 8.3.
2. **Shared node daemon** Group managers residing on the same node as the daemon are allocated to separate JVMs, as illustrated by Node X2 in Figure 8.3.
3. **Shared site daemon** A set of group managers at a site share a daemon within the same site, but on different nodes as shown by Site Y in Figure 8.3.

The last two categories share two important characteristic properties. They both communicate by means of Java RMI. And the replicas (and their associated GMs) are created on nodes/JVMs that are separate from the daemon, enhancing failure independence between the application replicas. On the other hand, in the *Shared JVM daemon* category, communication between the GMs and the daemon is reduced to plain *Local Method Invocations* (LMI). Consequently avoiding the communication overhead of RMI. However, in this case the failure of a single application replica is likely to bring down all other replicas in that JVM, including the daemon.

Table 8.1: Statistics for the communication overhead of using RMI between the GM and the daemon (values are in milliseconds). The statistics were obtained from measurements of 1000 invocations of the ping() method (invoked once every second) on an unloaded system.

	Mean	StdDev	Max	Min
Shared JVM daemon	0.00302	0.00032	0.00460	0.00242
Shared node daemon	0.88315	0.33364	4.16303	0.74455
Shared site daemon	0.81369	1.26693	28.42181	0.58962

Table 8.1 presents a few simple statistics that illustrate the overhead of RMI communication between the GM and the daemon. As the results show, using a *Shared JVM daemon* significantly reduces the communication overhead between daemon and GM, since communication is by means of local method invocations. Moreover, the latency difference between *Shared site* and *Shared node* is only marginal, however the variance when communicating between nodes is much higher; in particular, note the max value of 28 ms for *Shared site*. These variations are likely due to external stochastic components such as buffer delays etc. Also note the slightly higher mean value for *Shared node*; this can be explained by exposure to context-switching delays since there are two JVM processes on the same node that need to be activated before concluding the measurement.

For the first two categories above, the replicas and their associated daemons share the same fate with respect to *node crash failures*. Therefore, the failure detection mechanism is implemented in the daemon, avoiding that every GM in the system is pinging each other. Failure detection issues are discussed further in Section 8.5.

For the *Shared site daemon* category, application replicas may reside on nodes that are separate from the daemon node, and thus these replicas may crash independently of the daemon. Furthermore, in the *Shared node daemon* category, each replica's JVM may crash independently of the other replicas, as long as they are hosted in JVMs separate from the daemon.

Note that although all three categories discussed above are supported, the *Shared site daemon* approach is not used in the experiments presented in Part IV. Instead, a combination of the first two are used, as shown in Figure 5.1, where the daemon is co-located with the factory JVM. This is simply to reduce the complexity of the experiment setup.

8.4 Daemon Discovery and Creation

In this section, the algorithm used to discover node-external daemons is presented. The algorithm is required for the *Shared site daemon* approach, which may be used to improve the scaling properties of a system. The system-specific scaling policy defines the number of Jgroup daemons d_x, that should be available at each site x.

When application replicas are created they use the getGroupManager() method to obtain a GM as discussed in Chapter 6, which is needed to access the group communication services provided by the GM protocol modules. Next, the GM must either connect to an existing daemon, or a new daemon must be created. Whether to create a new daemon or not depends on the scaling policy, i.e. if the number of available daemons are below the required d_x. Algorithm 8.2 is used to find daemons and select the least loaded daemon within the local site. By least loaded is meant the least number of GMs (members) connected to a daemon.

Algorithm 8.2 The daemon discovery algorithm.

```
 1: Initialization:
 2:   site ← Config.getLocalSite()                    {The local site}
 3:   d_x ← site.getPolicy(daemons)                   {Expected # of daemons in the local site}
 4:   minMembers ← MAXVALUE                           {Min. # of members; initially the max value}
 5:   bestDaemon ← null                               {The least loaded daemon}
 6: Daemon discoverDaemon()                           {FINDS DAEMON IN LOCAL SITE}
 7:   foreach node ∈ site
 8:     daemon ← node.lookup("DAEMON")                {Query the nodes BootstrapRegistry}
 9:     if daemon ≠ null
10:       if node.isLocal()
11:         return daemon                             {Always use a local daemon if exists}
12:       d_x ← d_x − 1                               {Found a daemon; decrement d_x}
13:       members ← daemon.members()                  {# of members connected to this daemon}
14:       if members < minMembers
15:         minMembers ← members                      {Found a daemon with with less members}
16:         bestDaemon ← daemon                       {Save least loaded daemon}
17:   if d_x > 0
18:     bestDaemon ← Daemon.createDaemon()            {Create new daemon on local node}
19:   return bestDaemon
```

The algorithm works as follows. In the initialization part, the desired number of *daemons* at the local site, i.e. the d_x value, is obtained from the system-specific policy and other state variables are initialized. Next the algorithm iterates over the set of nodes in the local site, and queries each node (lookup()) to see if there is a daemon reference stored in the node's BootstrapRegistry. For each daemon that is found, a

check is performed to see if it is a node-local daemon, in which case the daemon is returned immediately. Node-local daemons are always used. For non-local nodes running a daemon, d_x is decremented and the algorithm queries the daemon to get the current number of $members$ (GMs) connected to it. The algorithm ignores daemons with too many connected members, saving the daemon reference only if it is the least loaded daemon found so far. At the end of the algorithm, the total number of daemons found is checked (d_x), and if enough daemons exists within the local site, the least loaded daemon is returned. Otherwise, a new daemon is created on the local node.

8.5 Failure Detection and Recovery

Replicas on the same node share the same fate with respect to node crash failures. However, replica processes (JVMs) may crash independently of other replicas on the same node, as long as they are hosted in separate JVMs. Moreover, placing the daemon in a separate JVM from the GMs enhances the failure independence between the JVMs hosting application replicas and the core group communication services.

To support multiple GMs on the same node, the daemon maintains two distinct membership lists: one regarding local replicas (GMs), and another concerning remote daemons (and their associated replicas). The inter-daemon failure detection mechanism is covered in [27]. However, the daemon also needs to detect failures of GMs, to accurately reflect the membership. This failure detection mechanism is the topic of this section. This failure detection mechanism is node (or site) local and only between the daemon and the GMs. Hence, it saves the cost of all members pinging each other as would be the case with a one-to-one mapping between daemon and member.

Figure 8.4 illustrates two possible crash scenarios, one involving the crash of a replica JVM (GM), and the other the crash of the daemon JVM. The latter scenario is the more severe case, as it renders all connected GMs incapable of communicating with its peer group members that may reside on remote nodes, since they rely on the daemon for such communication. In the former case, the remaining replicas are still able to communicate with its peer group members through the daemon.

The daemon-group manager failure detection mechanism is based on a simple *leasing* technique [32]. The technique aims to detect:

- The failure of connected replicas (GMs).
- The failure of the Jgroup daemon itself.

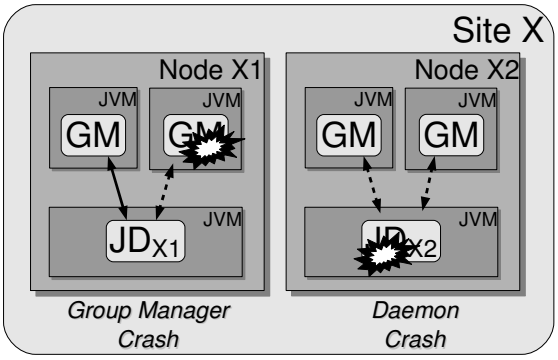


Figure 8.4: Two scenarios involving JVM crashes.

Algorithm 8.3 The failure detection algorithm (daemon side).

```
 1: Initialization:
 2:    members ← ∅                                  {The set of members associated with the daemon}
 3:  void connect(member)                           {INVOKED BY MEMBER TO CONNECT TO DAEMON}
 4:    members ← members ∪ member
 5:    member.timer.schedule(pingRate)
 6: when timer expires for member                   {PERIODIC PINGING}
 7:    member.hasFailed ← member.ping()             {Check if member is responding}
 8:    if member.hasFailed
 9:       members ← members − member                {Remove member from local member list}
10:       handleLocalSuspect(member)                {Runs view agreement protocol}
11:    else
12:       member.timer.reschedule(pingRate)         {Reschedule timer for this member}
```

That is, JVM crash failures are detected, and implicitly node crashes when the daemon or GMs are on remote nodes.

Algorithms 8.3 and 8.4 show the failure detection technique for the two entities involved. The technique works as follows:

- Algorithm 8.3: A replica first connects to the selected daemon (either found locally or on a remote node within the site). The daemon adds the $member$ to its list of $members$ and schedules the ping timer to be executed periodically with $pingRate$ intervals. Every time the timer expires, ping() is executed, and determines if the $member$ has failed or not. If is has failed, the $member$ is removed from the list of $members$, and the *view agreement protocol* is executed to form a new view, excluding the failed "local" member. Otherwise, the timer associated with the member is rescheduled.

Algorithm 8.4 The failure detection algorithm (group manager side).

```
1: Initialization:
2:    timer ← new TimerTask()                          {The daemon timer}
3: void ping()                                         {INVOKED PERIODICALLY BY DAEMON}
4:    timer.reschedule(pingRate × 2)
5: when timer expires                                  {GROUP MANAGER DETECT DAEMON FAILURE}
6:    System.halt()                                    {Daemon JVM has crashed; terminate replica}
```

- Algorithm 8.4: On the replica (group manager) side, the ping() method simply reschedules the local timer. Since the GM expects the ping() method to be invoked periodically (with rate $pingRate$) it can determine whether the daemon is late and should therefore be suspected to have crashed. To reduce the risk of false suspicion, a delay of $2 \times pingRate$ is allowed between each ping() invocation before suspecting the daemon and terminating itself.

8.5.1 Recovery Issues

As shown in Algorithm 8.3, when the daemon detects a failed member it invokes the handleLocalSuspect() method to force a run of the view agreement protocol. The details of the view agreement protocol is provided in [87]. The daemon does not attempt to recover a locally failed application replica. Instead the recovery is handled by the ARM framework as will be discussed in Chapter 11. Hence, it is imperative that replica failures are discovered by the daemon and that this information is communicated to other daemons, ultimately resulting in the installation of a new view excluding the failed member. The ARM framework use the view installations as the basis for activating recovery actions.

The more severe JD_{X2} failure shown in Figure 8.4 is also handled by ARM. The failure of the daemon is detected by other daemons by means of the inter-daemon failure detection [27], subsequently installing a new view that is used by ARM to take the appropriate countermeasures. Assuming that the daemon crashed, and not its connected replicas, Algorithm 8.4 running on the replica side will also detect the daemon failure. In response to detecting a daemon failure, the replicas simply commit suicide.

The reasons for not performing local recovery in either the daemon or the GM is complexity:

- Replicas that are incommunicado with the rest of the group through the daemon, may interfere with external entities, potentially causing inconsistencies.

- It would add significant complexity to ARM, as it would have to distinguish between failures being handled by the GM or the daemon from failures affecting nodes running daemon and GMs.
- Reestablishing daemon/GM state is difficult.

Chapter 9

Client-side Membership Issues in the Open Group Model

In a distributed fault-tolerant server system realized according to the open group model [64], inconsistency will (temporarily) arise between the dynamic membership of the replicated service and its client-side representation in the event of server failures and recoveries.

This chapter investigates the potential issues that may arise from such inconsistencies, and proposes techniques for maintaining client-side consistency and discusses their performance implications in failure/recovery scenarios where clients load balance requests on the servers. Moreover, client-side consistency is also related to the consistency of the dependable registry.

Comparative performance measurements have been carried out for two of the proposed techniques. The results are presented in Chapter 14. They indicate that the performance impact of temporary inconsistency is easily kept small, and that the cost of both techniques are small.

The chapter is based on [78] and is structured as follows. Section 9.1 motivates for the need to update the client-side membership. Section 9.2 discusses client-side performance impairments and the various delays involved in the update problem. Section 9.3 presents two techniques for maintaining the consistency of the dependable registry, whereas in Section 9.4 two proposed techniques for solving the client-side update problem are discussed.

9.1 Problem Description

In Jgroup, consistent server-side group membership is guaranteed through a group membership service, as discussed in Chapter 3. As discussed in Section 2.3.3, the *open* group model [64] enables external clients to interact transparently with the object group, as if it were a single, non-replicated server object. This is different from the *closed* group model, in which clients must become member of the group prior to any interaction with it, thus making it less scalable with respect to the number of simultaneous clients.

In order for clients to communicate with the object group, they need to obtain an object group reference (client-side group proxy). In Jgroup, this is accomplished using the dependable registry service [86], discussed in Section 3.4. For a replicated service it is common that this client-side group proxy holds information about the entire object group, allowing the client-side to perform transparent failover to a different group member should some member fail. However, the client-side view of the group membership is only an approximation of the server-side view, due to the open group model.

This chapter addresses issues concerning maintaining consistency between the dynamic server-side group membership and the client-side representation/approximation of that membership. In using the open group model we sacrifice the benefit of membership consistency inherent in the closed group model. However, assuming that the client-side group proxy holds enough live members to perform failover to another member, we are able tolerate some inconsistencies between the server and client-side membership. Unless the client-side membership is updated in some way however, the client will sooner or later become exposed to server-side failures.

Another aspect is keeping the naming service consistent with the server-side group membership. Building a dependable distributed middleware platform requires the naming service to be fault tolerant, so as to ensure that clients can always access the service. Both Jgroup [86] and Aroma [96] supports a dependable registry service, yet these do not update their database of client-side proxies in the presence of failures. Thus, even if the client application is exposed to server-side failures and is able to obtain a new client-side proxy from the registry service, this proxy may not reflect the correct membership of the group. The proxy will also contain references to failed servers, forcing clients to perform failover for the same server multiple times, leading to increased failover latency.

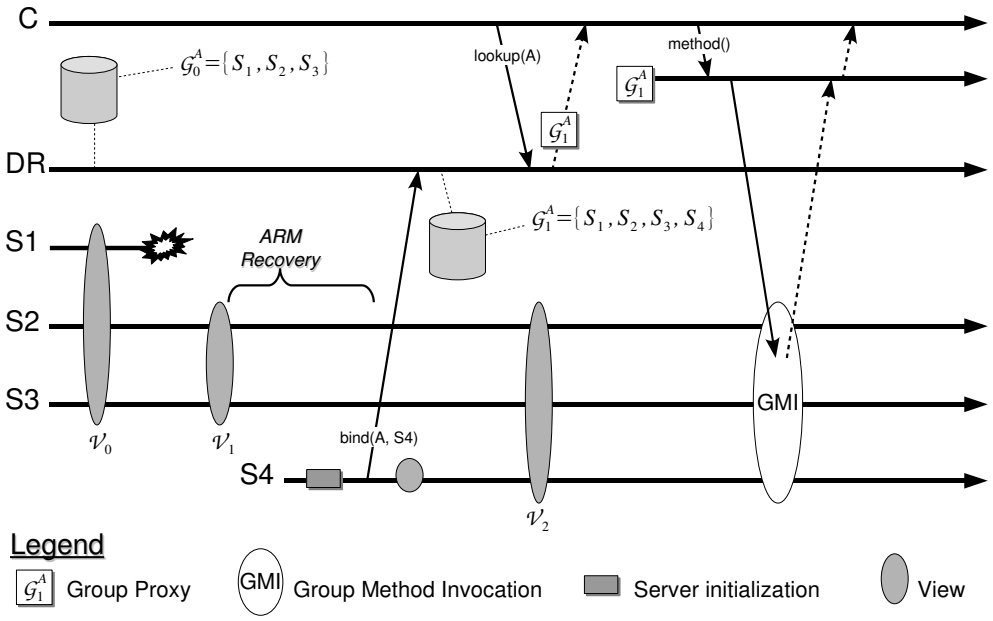


Figure 9.1: Failure/recovery scenario causing inconsistency in the registry.

Figure 9.1 shows the same failure/recovery scenario as in Figure 4.3, but serves to illustrate how this sequence of events will cause an inconsistency between the dependable registry and the server-side group membership. Initially, group A has a three member view $\mathcal{V}_0 = \{S_1, S_2, S_3\}$, and the dependable registry database holds a group proxy $\mathcal{G}_0^A$ which is identical to $\mathcal{V}_0$. After some unspecified time S_1 crashes. This leads to the installation of a new view, $\mathcal{V}_1 = \{S_2, S_3\}$, after which ARM is notified of the failure (not shown in Figure 9.1). ARM attempts to maintain the redundancy level by installing a replacement replica, S_4, consequently installing view $\mathcal{V}_2 = \{S_2, S_3, S_4\}$. During initialization of S_4, bind() is invoked on the registry in order to associate the S_4 group member with the $\mathcal{G}^A$ client-side proxy.

Later, once a client needs to access group A, it contacts the registry to obtain the client-side proxy. Given $\mathcal{G}^A$, the client can perform invocations on all live members of the group. Note that S_1 still remains in the registry database, even though it has crashed. This is since there is no update mechanism in place, and hence there will be a persistent inconsistency between the server-side view $\mathcal{V}_2$ and the entry $\mathcal{G}_1^A$ in the registry. Evidently this inconsistency propagates to clients, as seen in Figure 9.1.

9.2 Client-side Performance Impairments

9.2.1 Performance Without Updates of the Client-side Proxy

To demonstrate the performance penalty of not updating the client-side membership in accordance with the dynamic server-side group membership, we have performed several experiments on a four server system with crashes and recoveries in which the clients did not update their membership. The clients perform load balanced invocations on all known servers, using the *anycast* method semantic. The method invoked takes a 1000 byte array as argument, and returns the same array back to the client.

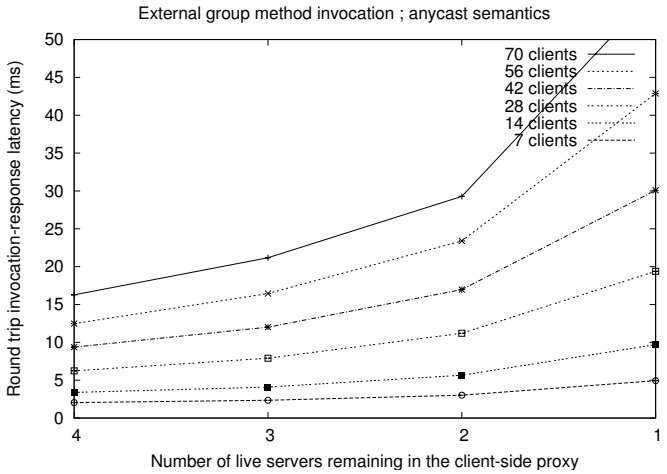


Figure 9.2: The performance drop due to not updating the client-side membership.

Figure 9.2 shows the results of the experiment. The plot shows several lines for various client loads ranging from 7 to 70 clients. Only 7 physical machines are used for the clients, whereas in *all cases* the four servers run on dedicated machines, i.e. only the number of live servers remaining in the client-side proxy varies. Initially, all client-side proxies contained all four servers. During the experiment servers crashed and recovered, rendering the client-side proxies inconsistent (having fewer live servers). Not surprisingly the results show that the client-side proxies should update their membership information to avoid increased invocation latencies. Once a client detects a server failure, it is removed from the client-side proxy and not replaced by new servers when they are installed. Thus, the observed performance drop

is due to contention at the servers, since the load balancing mechanism in the client-side proxy does not know all the servers.

Another, perhaps more important problem is the scenario where all servers crash before updating the proxy, rendering failed servers visible to clients. Figure 9.3 shows such a scenario, where the first client invocation completes and the second invocation fails due to all servers having crashed and recovered as new instances, and since the client-side proxy $\mathcal{G}^A$ does not know any of the new server instances it is not able to failover. Hence, the invocation fails and the client is exposed to the failure.

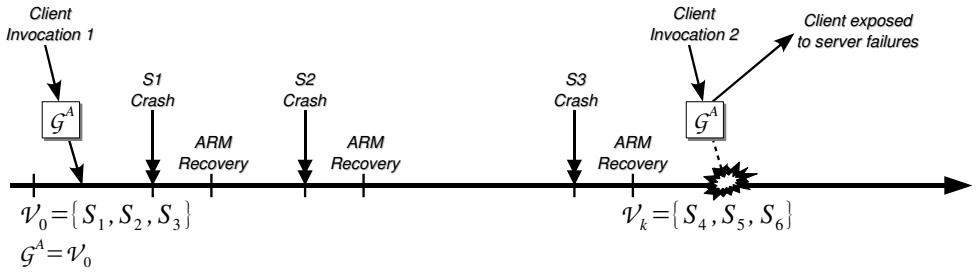


Figure 9.3: All servers crashed, exposing the failure to the client.

9.2.2 Client-side Update Delays

To be able to reason about the timing involved in updating the client-side proxy ($\mathcal{G}^A$), let's assume that the proxy is updated in some unspecified manner. The timeline in Figure 9.4 illustrates one possible failure/recovery scenario in which the client-side group proxy is updated. In this example a group of three members install a view $\mathcal{V}_0 = \{S_1, S_2, S_3\}$, and bind their remote references in the registry service, allowing clients to obtain a group reference (the $\mathcal{G}^A$) from the registry. After some unspecified time S_1 crashes, rendering all existing $\mathcal{G}^A$ instances inconsistent with the actual situation. Given that there are two remaining servers in the group, the $\mathcal{G}^A$ can simply failover to another server, to perform a client invocation.

The $\mathcal{G}^A$ should however be updated to enable the use of all available servers, and the timeline in Figure 9.4 illustrates the timing involved in updating the $\mathcal{G}^A$. Let t_0 denote the time at which S_1 crashes. Let $T_{\mathcal{V}_1}$ be the time that it takes for the other servers to detect the failure and agree on a new view ($\mathcal{V}_1$), while $T_{\mathcal{V}_2}$ is the time it takes to install a replacement server (S_4) and for the servers to agree on the new view $\mathcal{V}_2$. Thus, the

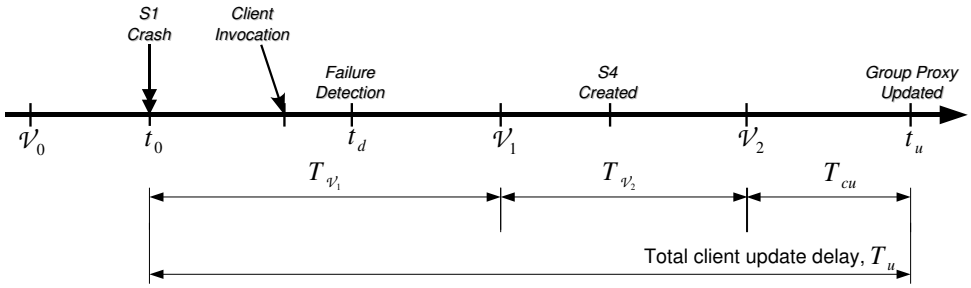


Figure 9.4: The update timeline.

sum corresponds to the (server-side) recovery time, $T_r = T_{\mathcal{V}_1} + T_{\mathcal{V}_2}$. Furthermore, let t_d be the time that the client-side proxy detects that S_1 has crashed. The total time for updating $\mathcal{G}^A$ is given by $T_u = t_u - t_0$, and we denote this time as the *total update delay*. Note that t_d may stretch beyond t_u, when $\mathcal{G}^A$ does not select to use S_1, or if the client does not perform any invocations in the range $[t_0, t_u]$. Finally, let the *client update time*, T_{cu}, be the time from the installation of the compensation view $\mathcal{V}_2$ and until $\mathcal{G}^A$ is again consistent with the actual situation.

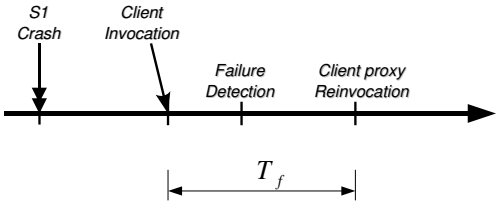


Figure 9.5: The failover latency.

The failover latency, T_f, is the additional time imposed on clients when attempting to invoke a server that has failed. Let T_f be the time between the proxy receiving the actual invocation and the time of performing a reinvocation on a different server, as illustrated in Figure 9.5. The failover latency does not include the actual invocation-response latency. Generally, it is assumed that failover requires only a single reinvocation. However, for multiple nearly coincident failures, or if the update delay is long, the failover latency may involve multiple invocation attempts.

In the following, extensions to the client-side proxy mechanism of Jgroup/ARM, and its dependable registry service [86] are presented. The purpose of these extensions is

to maintain consistency between the server-side group membership and its representations both at the client-side and in the registry service database.

9.3 Updating the Dependable Registry

The current implementation of the dependable registry described in Section 3.4 lacks support for updating its content to ensure consistency with the server-side group membership. To better understand why this is a problem, consider the following scenarios: (i) a server leaves the group voluntarily; and (ii) a server leaves the group involuntarily by crashing or partitioning. In the former case, the server may perform the unbind() method on the registry, allowing the registry database to be updated accordingly, by removing the server's reference from $\mathcal{G}^A$. In the latter case, however, the failed server is unlikely to be able to perform the unbind() method. The registry database is thus rendered inconsistent with respect to the server-side membership $\mathcal{V}$. In this situation, the dependable registry will continue to supply clients with a proxy ($\mathcal{G}^A$) that contains servers that are no longer member of the group. In fact, the proxy may be completely obsolete if all servers in the group have crashed. Furthermore, if new servers were to be started to replace failed once as is done by the ARM framework, the number of members of the group proxy for $\mathcal{G}^A$ would grow to become quite large. Figure 9.1 illustrates the problem visually.

To prevent clients from obtaining obsolete proxies from the dependable registry, two distinct techniques are provided for maintaining consistency of the registry content. The techniques are implemented as separate protocol modules embedded within the group manager associated with servers. A detailed description of the techniques can be found in [79]. Note that similar techniques are also provided by the ARM framework for recovery purposes. Hence, reusing the techniques from ARM would improve efficiency. However, it would also require tighter integration between the replication manager and the dependable registry, sacrificing modularity and independence of the dependable registry.

9.3.1 The Lease Refresh Technique

Our first solution to the problem is based on the well known concept of leasing [32]. By leasing we mean that each server's object reference as stored in the dependable registry is only valid for a given amount of time called the $leaseTime$. When a

server's object reference has been in the registry for a longer period of time than its $leaseTime$, it becomes a candidate for removal from the registry database. To prevent such removal, the server must periodically renew its lease with the registry. The interval between these refresh() invocations is referred to as the $refreshRate$, typically related by a factor of two as follows: $leaseTime = 2 \times refreshRate$. Figure 9.6 illustrates the workings of the LeaseModule.

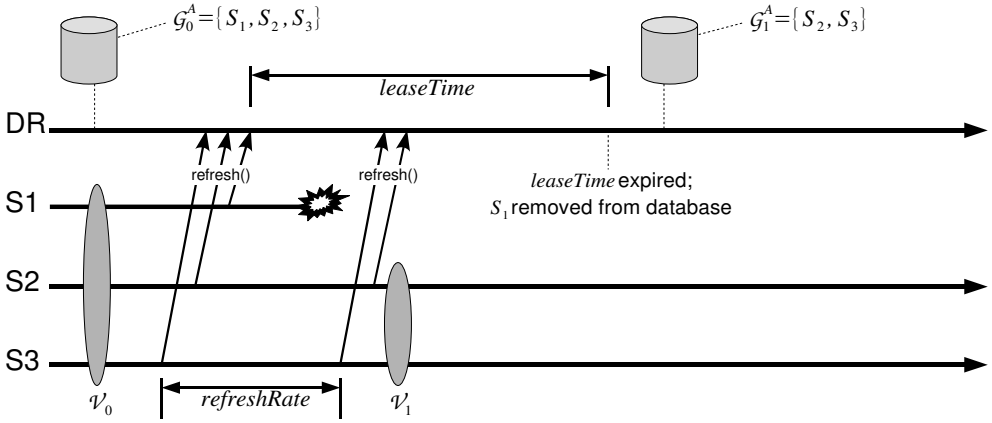


Figure 9.6: The LeaseModule exemplified. S_1 has crashed and consequently it is removed from the registry since its lease is not renewed.

This approach is also used by the Jini lookup service [7], except for the fact that it can only associate a single server with each service name.

9.3.2 The Notification Technique

Jgroup provides a group membership service that allows servers (or modules) to receive notification of changes to the group's composition. These notifications come in the form of viewChange() method invocations. Thus, upon receiving such a view change event, the NotifyModule selects a leader (e.g. S_3) for the group. The leader is responsible for updating the dependable registry in case the new view represents a contraction of the group's membership. This is done by executing the unbind() method (with S_1 as argument) on the registry. Figure 9.7 illustrates the workings of the NotifyModule.

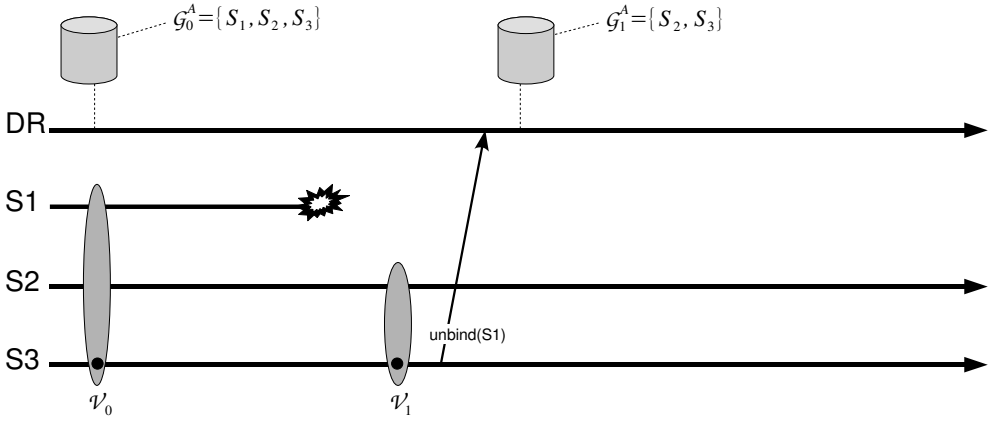


Figure 9.7: The NotifyModule exemplified. S_3 is the leader.

9.3.3 Combining Notification and Leasing

The NotifyModule is by far the most interesting and elegant technique, but it has a drawback in situations when there is only one remaining server in the object group. In this case the last server will be unable to notify the registry when it fails. However, it is easy to combine the LeaseModule and the NotifyModule in order to exploit the advantages of both. Let $|\mathcal{V}|$ denote the size of the current view $\mathcal{V}$ installed by the group. The default for the combined approach is to use the NotifyModule (i.e. when $|\mathcal{V}| > 1$), and in the situation with only one (i.e. when $|\mathcal{V}| = 1$) remaining server the LeaseModule is activated. In doing so, we will also diminish the main drawback of leasing technique, namely the amount of generated network traffic, since there is only one server that needs to perform a refresh() periodically.

9.4 Updating the Client-side Proxies

Even though the dependable registry is kept up-to-date using the aforementioned techniques, the client-side proxy representation of the group membership will still become (partially) invalid since it is not updated in any way. Since the membership information known to the client-side proxy may include both failed and working servers, the proxy may hide that some servers have failed by using those that work. For each *anycast* invocation, the client proxy randomly selects a single server among the working servers.

In previous versions of Jgroup, the client-side proxy throws an exception to the client application if all group members have become unavailable, rendering server failures visible to the client application. The solution is simple enough; the client application can contact the registry in order to obtain a fresh copy of the group proxy, assuming the registry is updated. However, this yields a poor load distribution among the servers in the period until obtaining a fresh copy from the registry. Moreover, the operation should also be transparent to the client application programmer. Hence, two techniques that can be used to obtain such failure transparency are proposed.

1. **Periodic refresh** The client-side proxy requests a new group proxy from the dependable registry at periodic intervals.
2. **Client-side view refresh** For each invocation, the client-side proxy attaches its current view identifier v_k. The server compares the client view with its own view. If the two differs, the server augments the result message with its current view, allowing the client to update its membership information immediately after the invocation. Figure 9.8 illustrates the approach.

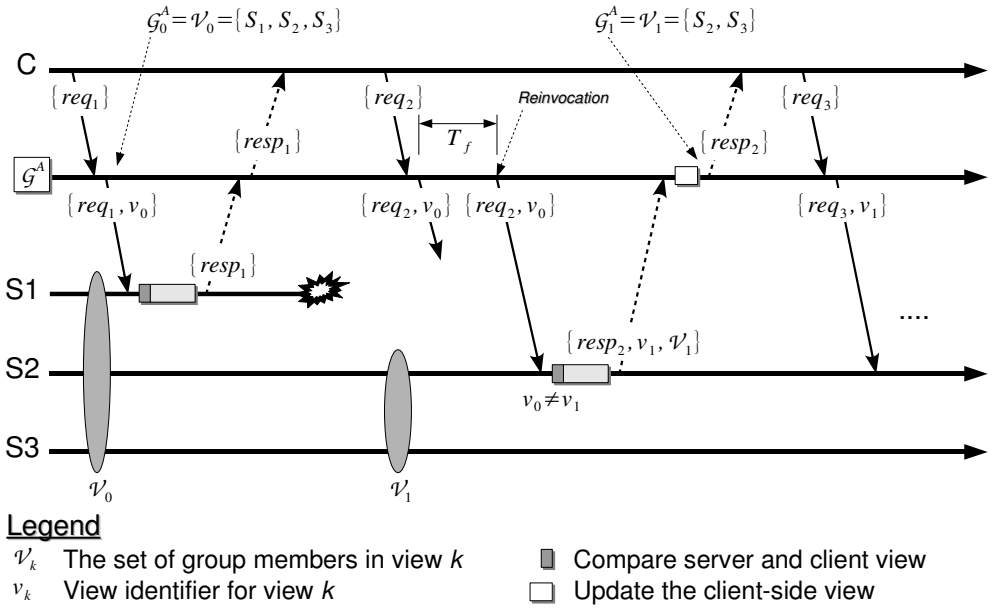


Figure 9.8: The client-side view refresh technique.

Technique 1 requires selecting a suitable refresh rate interval, which can be difficult. If set too low, it could potentially generate a lot of overhead network traffic, and if

set too high we run the risk that it is not updated often enough to avoid exposing server failures to the client. This technique works by indirect updates in that it relies on the registry already being up-to-date, which may not be the case, depending on the technique used to update the registry. Therefore, Technique 2 is very appealing in that it will work even though the registry is not updated, since the server-side handles updating clients itself. The overhead of the technique is also very small; only a 64-bit view identifier is added to each invocation performed, and a simple comparison is performed on the server-side. In response to an invocation whose client-side view (e.g. v_0) differs from the server-side view (e.g. v_1), the reply message is simply augmented with the current server-side view $\mathcal{V}_1$ and its identifier v_1.

The main difference between these techniques is the time it takes the client-side proxy to return to a consistent state with respect to the membership of the server group. It is not an issue concerning server availability, but rather the ability of the client-side proxy to load balance its invocations on all active servers, and not just the ones that are known to the clients. Furthermore, Technique 2 may also experience less failovers, since it is possible that the proxy is updated without even noticing a server failure. With the periodic refresh technique it is more likely that the proxy have to perform failover.

Both techniques discussed above can be combined with a mechanism to circumvent the problem illustrated in Figure 9.3. Such a client-side mechanism would simply query the registry if the set of known servers have been exhausted to become updated again. Obviously, the mechanism assumes that the registry is also kept up-to-date using the techniques in Section 9.3.

All techniques described in this chapter have been implemented. However, only the client-side view refresh technique is enabled by default. Chapter 14 presents measurement results and a comparative evaluation of the two techniques discussed above.

Part III

Autonomous Management

Chapter 10

Policies for Replication Management

The ARM framework, introduced briefly in Chapter 4, aims to be a fault treatment architecture that is self-managing and adaptive to node and network dynamics and changing application requirements. In general, an autonomous system seeks to limit the number of manual interactions needed, and avoids direct manipulation of system components for management purposes. One approach to accomplish this is to use a policy-based management system [113]. Policy-based management enables system administrators to specify how a system should react to changes in the environment — with no human intervention. These specifications are called *policies*, and describes how the system should be managed under various dynamically changing system conditions.

Policy-based management architectures are often organized using two key abstractions [113, 2]:

- A (centralized) manager component, which monitors and analyzes the status of managed resources and plans and executes actions on the managed resources. This component is considered the policy decision point (PDP) [2].
- Managed resources which are controlled by the manager component. Managed resources can be considered the policy enforcement point (PEP) [2].

Typically, the managed resources expose interfaces for sensors and effectors which are used by the manager component [2].

The purpose of this chapter is to describe the policy and configuration mechanism used by Jgroup/ARM to determine the self-managing behavior of the system. The general organization of the ARM architecture (see Chapter 4) is similar to that of a manager component and associated managed resources [2]. However, ARM policies are not restricted to a centralized PDP, e.g. consider the decentralized remove policy discussed in Section 10.1.3.

Three distinct policies are used to guide ARM in making decisions on how to handle a particular system condition, e.g. a replica failure. Overall, these policies enforce the dependability requirements and WAN partition robustness of services. The three policy types are complemented with configuration information specifying required policy input. The policy examples used to guide ARM behavior are simple and aimed to demonstrate the concept; the policies may be extended or combined with other policies to obtain more advanced behaviors, e.g. to consider the system load when making policy decisions.

The policy framework for Jgroup/ARM is designed as an integral part of ARM, and hence is not intended to be a generic policy framework comparable to e.g. [2]. In particular, there is no conflict resolution mechanism for ARM policies.

The three policy types are presented in Section 10.1. Section 10.2 discusses the configuration mechanism used by the policies to obtain configurable input data.

10.1 ARM Policies and Policy Enforcing Methods

Policy-based management allows a system administrator to provide a high-level policy specification to guide the behavior of the underlying system. ARM requires that three separate policy types be defined in order to support the autonomy properties:

1. The *distribution policy* which is specific to each ARM deployment.
2. The *replication policy* which is specific to each service deployed through ARM.
3. The *remove policy* which is specific to each service deployed through ARM.

10.1.1 The Distribution Policy

The purpose of a distribution policy is to compute how service replicas should be allocated onto the set of available sites and nodes in the target environment (see Figure 1.1). Generally, two types of input are needed to compute the replica allocations of a service:

1. the target environment, and
2. the number of replicas to be allocated.

The latter is obtained at runtime from the replication policy associated with the service being deployed, whereas the former is typically obtained from the target environment configuration as exemplified by Listing 10.1 on page 142.

Currently, ARM supports only one distribution policy (`DisperseOnSites`) that will avoid co-locating two replicas of the same service on the same node, while at the same time trying to disperse the replicas evenly on the available sites. In addition, it will try to keep the replica count per node to a minimum. The same node may host multiple distinct service types. It is assumed that WAN connections between sites are less robust than LAN connections. Hence, the objective of this distribution policy is to ensure available replicas in each *likely* network partition that may arise. More advanced distribution policies may be defined by combining the above policy with load balancing mechanisms.

Table 10.1: High-level abstractions for replica distribution.

Method name	Description
assignReplicas()	Obtain node allocations for the given service.
removeReplicas()	Remove node allocations for the given service.
reassignReplica()	Relocate a replica to a new node.
colocateReplicas()	Co-locate replicas on the same set of nodes.
getAssignedNodes()	Get the current node allocations for the given service.
notify()	Event notification handler.

A new distribution policy is defined through a set of high-level abstractions for replica distribution, as shown in Table 10.1. Hence, a replica distribution algorithm must implement those abstractions, allowing the replication manager to install replicas based on the output (node allocations) generated by the distribution algorithm. To make node allocation decisions, a distribution algorithm may wish to receive ARM events, such as view change events received from external groups. This is possible through the notify() abstraction.

10.1.2 The Replication Policy

Each service that is to be deployed through ARM is associated with a replication policy. The primary purpose of the replication policy of a service is to describe how the redundancy level of that service should be maintained. Generally, two types of input are needed:

1. The target environment.
2. The initial/minimal redundancy level of the service.

Let R_{init} and R_{min} denote the initial and minimal redundancy levels. Currently, only one replication policy (`KeepMinimalInPartition`) is provided whose objective is to maintain service availability in all partitions. That is to maintain R_{min} in each partition that may arise. To maintain a given redundancy level, a fault treatment (recovery) action such as installing replacement replicas is typical. Alternative policies can easily be defined, for example to maintain R_{min} in a primary partition only. Or a policy may interpret group failures as a design fault symptom and revert to a previous implementation of the service if such exists. Such a policy could be useful in conjunction with the online upgrade technique presented in Chapter 12.

As discussed above, a replication policy describes how recovery from various failure scenarios should be handled for its associated service. The replication policy is defined through a set of high-level abstractions for replica recovery, as illustrated in Table 10.2.

Table 10.2: High-level abstractions for the replication policy.

Method name	Description
needsRecovery()	Check if the service needs recovery.
prepareRecovery()	Prepare to perform recovery.
handleFailure()	Perform recovery action.

When deploying a service, ARM will instantiate a service-specific replication policy. During its operation, ARM receives events and maintains state associated with each of the deployed services, including updating the redundancy level of services. This state is used by the replication policy to determine the need for recovery. If ARM detects that it needs to perform recovery, it first initializes state variables through

the prepareRecovery() abstraction and then invokes the handleFailure() abstraction. Internally, a replication policy needs to analyze the failure pattern to determine the fault treatment action to use. Table 10.3 lists three predefined fault treatment actions that can be reused.

Table 10.3: Common fault treatment actions used by a replication policy.

Method name	Description
restartReplica()	Restart a service replica on a given node.
relocateReplica()	Relocate a service replica away from a given node.
groupFailure()	Called when all service replicas have failed.

10.1.3 The Remove Policy

Each service deployed through ARM is associated with a remove policy. The remove policy of a service describes how the redundancy level should be adjusted if the redundancy level exceeds some threshold, e.g. the initial redundancy level, R_{init}, of the service. As opposed to the replication policy described above, the remove policy is executed in a decentralized manner without involving the replication manager. That is, it is the supervision modules associated with each of the replicas which are collectively responsible for deciding which replica should be removed when the redundancy level exceeds the threshold. As the remove policy is embedded in the supervision module, further details are given in Section 11.3.3.

10.2 The Configuration Mechanism

Policy specifications obtain input from a simple configuration mechanism, based on XML, that enables administrators to specify (1) the target environment, (2) deployment-specific configuration parameters, and (3) service-specific descriptors. In the following two sections, the target environment configuration and service configuration formats are discussed.

10.2.1 Target Environment Configuration

Listing 10.1 shows the format for the target environment configuration. It specifies the set of sites and the IP multicast address to be used internally within each site.

Listing 10.1: A sample target environment configuration.

```
<TargetEnvironment>
  <Site name="ux.uis.no" address="226.1.2.1">
    <ScalingPolicy daemons="all"/>
    <Node name="badne" port="51000"/>
    <Node name="johanna" port="51000"/>
  </Site>

  <Site name="item.ntnu.no" address="226.1.2.2">
    <ScalingPolicy daemons="1"/>
    <Node name="samson" port="54300"/>
    <Node name="poker" port="54300"/>
  </Site>

  <Site name="cs.unibo.it" address="226.1.2.3">
    <ScalingPolicy daemons="2"/>
    <Node name="annina" port="20100"/>
    <Node name="leonora" port="20100"/>
    <Node name="lily" port="20100"/>
  </Site>
</TargetEnvironment>
```

The name of a site typically corresponds to a DNS domain name. Specifying the target environment in this manner is required since there is no automated discovery mechanism that is able to span across multiple Internet sites (domains).

Note that a scaling policy is also associated with each site. It is used to configure the number of daemon instances within each site, as discussed in Chapter 8. Let d_x denote the number of daemons in site x, and let N_x denote the number of nodes in site x. Each site x should specify the value $d_x \in [1, N_x]$. If the keyword `all` is specified, or if the scaling policy is undefined for site x, then $d_x := N_x$ is used.

Each site entry also defines its associated set of nodes. A node entry specifies its DNS host name (or IP address), and the port number through which communication should occur. Other properties could also be added to the node entry, such as the maximum number of replicas that the node should host.

Once the target environment configuration has been read, it is compiled into a dynamic runtime representation, allowing the system to reconfigure its set of nodes on-demand.

In general, only server-side applications need to have complete knowledge of the target environment. However, clients need to know at least the nodes that are hosting

the dependable registry service. This is necessary to obtain the client-side group proxies needed to communicate with other deployed services. In practice however, the client-side uses the same target environment configuration file.

10.2.2 Service Configuration

Listing 10.2 shows two service configurations, one describing the configuration of the ARM replication manager service and the other describing the replicated service used in the experiments in Part IV.

The service configuration permits the operator to define and specify numerous attributes to be associated with the service. Some of the attributes that can be specified include: The *service name* and implementation *class*, the *replication policy* to be used in case of failures, and the set of *protocol modules* used by the service. The *redundancy* levels to be used initially and to be maintained (minimal) are configurable parameters to the replication policy. In addition to these fixed parameters, the service configuration may define generic parameters that are both service-specific as well as module-specific. Typically, generic parameters are used to specify timeout values or boolean properties that enable a particular function, e.g. Lines 5-6 in Listing 10.2.

Prior to installation of a service, its service configuration is compiled into a dynamic runtime representation and passed to the replication manager. The replication manager maintains a table of deployed services and their corresponding runtime configurations, allowing the configuration of a service to be modified at runtime. This is useful to adapt to changes in the environment.

As mentioned in Section 10.1, the distribution policy is specific to each ARM deployment, and hence it is specified as a configuration parameter in the service configuration of the ARM replication manager, as shown in Line 15 in Listing 10.2. Note that the choice of replication protocol is not configurable through the service configuration, since this choice is highly dependent on the design of the service. Hence, instead the replication protocol is specified through annotation of individual methods in the server implementation as discussed in Chapter 7.

Listing 10.2: A sample service configuration description for the replication manager.

```
<Service name="ARM/ReplicationManager" group="2">
  <Class name="jgroup.arm.ReplicaManagerImpl"/>
  <ProtocolModules>
    <Module name="Supervision">
      <Param name="GroupFailureSupport" value="no"/>
      <Param name="RemoveDelay" value="5"/>
    </Module>
    <Module name="Registry"/>
    <Module name="EGMI"/>
    <Module name="StateMerge"/>
    <Module name="Multicast"/>
    <Module name="Membership"/>
    <Module name="Dispatcher"/>
  </ProtocolModules>
  <DistributionPolicy name="DisperseOnSites"/>
  <ReplicationPolicy name="KeepMinimalInPartition">
    <Redundancy initial="3" minimal="2"/>
    <ServiceMonitor expiration="3"/>
  </ReplicationPolicy>
</Service>

<Service name="ARM/ReplicatedService" group="200">
  <Class name="jgroup.test.ReplicatedServer"/>
  <Param name="SharedJVM" value="no"/>
  <ProtocolModules>
    <Module name="Supervision">
      <Param name="GroupFailureSupport" value="yes"/>
      <Param name="RenewalRate" value="30"/>
      <Param name="RemoveDelay" value="5"/>
    </Module>
    <Module name="Registry"/>
    <Module name="EGMI"/>
    <Module name="StateMerge"/>
    <Module name="Multicast"/>
    <Module name="Membership"/>
    <Module name="Dispatcher"/>
  </ProtocolModules>
  <ReplicationPolicy name="KeepMinimalInPartition">
    <Redundancy initial="3" minimal="2"/>
    <ServiceMonitor expiration="3"/>
  </ReplicationPolicy>
</Service>
```

Chapter 11

Autonomous Replication Management

Fault tolerant systems are able to continue to provide service in spite of the occurrence and activation of faults in the system. Redundancy is a common approach to build fault tolerant systems whereby faults are detected and their effects masked from clients using the system. Yet the dependability characteristics of fault tolerant systems based on redundancy can be improved further by the introduction of a fault treatment mechanism. A fault treatment system is one that is able to reconfigure the system to either rectify or reduce the consequences of a fault/failure.

The *Autonomous Replication Management* (ARM) framework is a self-managing fault treatment architecture that is adaptive to network dynamics and changing requirements. It was introduced briefly in Chapter 4, and in this chapter additional details are provided.

The aim of using ARM is to ultimately reduce the human interactions required to maintain the redundancy level of services, consequently improving the dependability characteristics of services deployed through ARM. Fault treatment is accomplished through a reactive mechanism in which failures are detected, followed by system reconfiguration, such as installing replacement replicas to restore the desired level of redundancy. The objective is to minimize the period of reduced failure resilience, in which additional failures could cause service delivery to stop.

Currently, ARM handles *object*, *node* and *network partition* failures. Both Delta-4 [102] and AQuA [104] handles *value* faults, but do not support network partitioning.

ARM is also able to handle multiple concurrent failure activities, including failures affecting the ARM infrastructure.

ARM also features a convenient mechanism to deploy services without having to worry about the distribution of replicas in the target environment; depending on the distribution policy, replicas are placed on nodes within different sites so as to reduce the likelihood of clients becoming partitioned from all replicas.

A non-intrusive system design is applied, where the operation of deployed services is completely decoupled from ARM during normal operation in serving clients. Hence, the overhead due to the main recovery mechanism is negligible.

The chapter is organized as follows: Section 11.1 describes the main ARM infrastructure component, the replication manager and its interfaces. Section 11.2 discusses the management client used to deploy services. Section 11.3 covers the protocol module that must be included in the protocol set of servers for which ARM should perform fault treatment. In Section 11.4 the object factory used to install and remove replicas is discussed, and in Section 11.5 the ARM failure recovery mechanism is discussed. Section 11.6 discusses issues of replicating the replication manager. Finally, Section 11.7 summarizes the benefits of the ARM framework.

11.1 The Replication Manager

The *replication manager* (RM) is the main component of the ARM infrastructure; its tasks are:

1. to provide interfaces for installing, removing and updating services;
2. to distribute replicas in the target environment, to (best) meet the operational policies for all services (see Chapter 10);
3. to collect and analyze information about failures, and
4. to recover from them according to the policies defined for the services.

The replication manager is designed as a central controller, enabling consistent decisions on replica placement and recovery actions. For increased fault-tolerance, however, it is replicated using Jgroup and exploits its own facilities for self-recovery and to bootstrap itself onto nodes in the target environment (see Section 11.6).

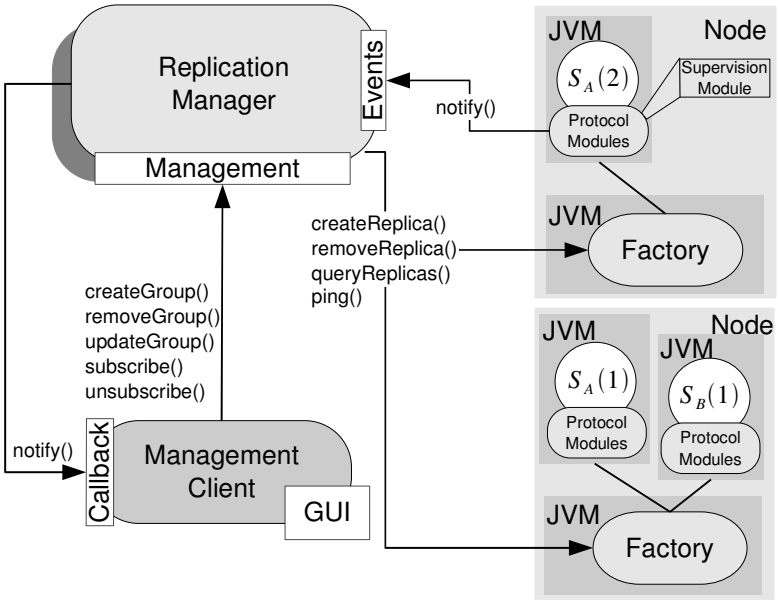


Figure 11.1: Overview of ARM components and interfaces.

Being based on the open group model adopted by Jgroup, external entities are able to communicate requests and events to the RM without the need to join its group, avoiding delays and scalability problems inherent to the closed group model [64].

Figure 11.1 is the same as Figure 4.4; it is repeated here for convenience. It illustrates the core components and interfaces of the ARM infrastructure.

Two EGMI interfaces are used to communicate with the RM. The Management interface is used by the management client to request group creation, update and removal. The Events interface is used by external components to provide the RM with relevant events for performing its operations.

As the Management interface is an EGMI interface, it means that on rare occasions, the same invocation request may be completed in concurrent views, causing ARM to create a service multiple times. This is simply a manifestation of the fact that "exactly-once" operation semantic is impossible to guarantee in the presence of failures [94]; see Section 3.3.2.2 for additional details. The supervision module, described in Section 11.3, provides measures to deal with this problem.

A collection of event types are supported, as shown in Table 11.1. All event types provide their own handle() method, which is executed by the RM upon receiving an

Table 11.1: Event types supported by ARM.

Event type	Description
ViewChange	Notify ARM of a change in the group composition of a service.
IamAlive	Renew the lease of a replica with the RM.
ReplicaFailure	Notify that a replica has failed.
NodePresence	Lets a node notify its presence in the target environment.

event. Events interact with the RM through a set of well-defined interfaces. This event handling architecture makes it very easy to augment the system with additional event types, assuming their interactions with the RM internals are limited to the methods supported by the well-defined interfaces. Some of these events are described further in Section 11.3 and in Section 11.4.

11.2 The Management Client

The management client enables a system administrator to install or remove services on demand. The management client may also perform runtime updates of the configuration of a service. Currently, updates are restricted to changing the redundancy level attributes. It is foreseen that the ability to update the service configuration can be exploited by ARM to support some degree of self-optimization.

Additionally, the management client may subscribe to events associated with one or more object groups deployed through ARM. These events are passed on to the management client through the Callback interface, permitting appropriate feedback to the system administrator. A management client may disconnect and later reconnect to the RM, and re-subscribe to callback events of previously deployed object groups.

The current management client implementation is specialized and supports defining scripts for automated installations. It is discussed further in Chapter 13, and was used to perform the experimental evaluations in Part IV. An alternative implementation may support a graphical front-end to ease human interaction with ARM.

11.3 Monitoring and Controlling Services

Keeping track of and controlling service replicas is essential to enable discovery of failures and to rectify any deviation from the specified dependability requirements.

Figure 11.2 illustrates the ARM failure monitoring architecture, in which a combination of mechanisms are provided. The architecture primarily follows an *event-driven* design in that external components report events collectively to the RM, instead of the RM continuously probing each individual component, which would be costly.

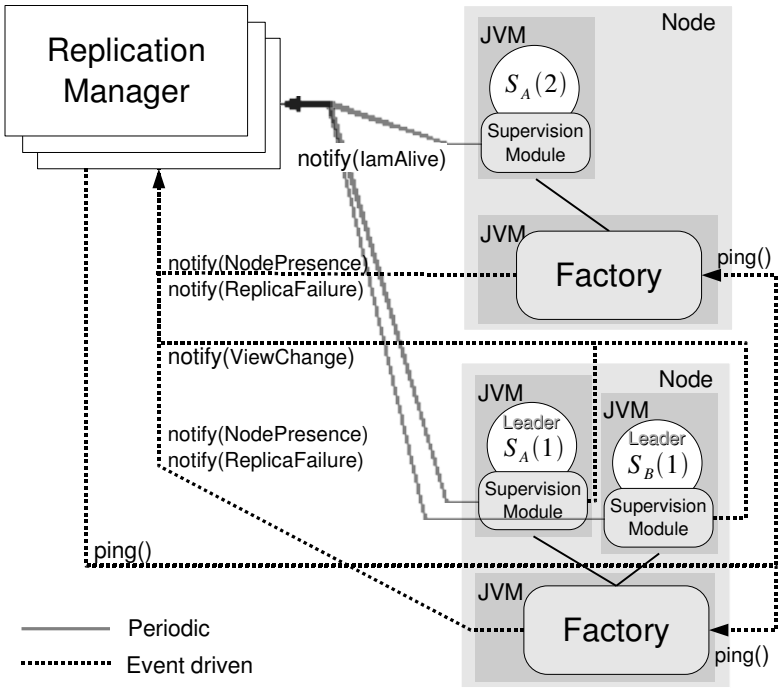


Figure 11.2: The ARM Failure Monitoring architecture.

11.3.1 Group Level Service Monitoring

To exploit synergies with existing Jgroup components, tracking services is performed at two levels of granularity: groups and replicas. Both tracking mechanisms are managed by supervision modules, that must be included in the set of protocol modules associated with Jgroup replicas.

At the group level, the leader of a group is responsible for notifying the RM of any variation in the group membership. In this way, the failure detection costs incurred by the PGMS are shared with the failure monitoring part of the RM.

View installations generated by the PGMS are intercepted by the supervision module and reported to the RM through ViewChange events. To avoid that all members of

a group report the same information, only the *group leader* (see Figure 11.2 and Figure 11.3) sends this information to the RM. Based on this information, the RM determines the need for recovery, as discussed in Section 11.5.

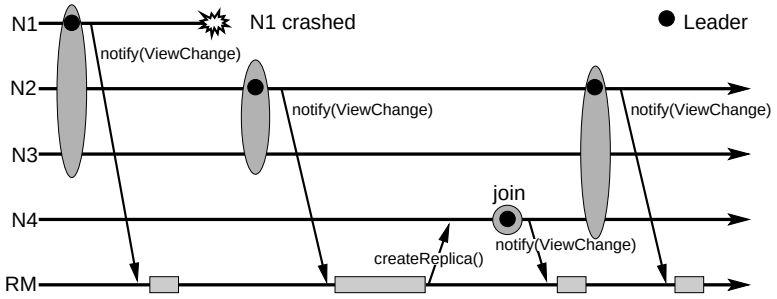


Figure 11.3: An example failure-recovery sequence (same as Figure 4.3).

As an illustration of the workings of the supervision module, Figure 11.3 shows a common failure-recovery sequence where node $N1$ fails, followed by a recovery action causing the RM to install a replacement replica on node $N4$.

11.3.2 Replica Level Monitoring

Unfortunately, group-level events are not sufficient to cover *group failure* scenarios in which all remaining replicas fail before being able to report a view change to RM. This can occur if multiple nodes/replicas fail in rapid succession, or if the network partitions, e.g. leaving only one replica in a partition whom fails shortly thereafter. For this reason, replica level tracking is also needed.

To handle group failures, a lease renewal mechanism [32] is embedded in the supervision module, causing all replicas to issue renew (IamAlive) events periodically to prevent ARM from triggering recovery, as illustrated in Figure 11.4. If an expected renew event is not received, ARM will activate recovery.

The rationale behind this technique is the assumption that group failures are extremely rare and typically become even less likely for larger groups. Therefore, as illustrated in Figure 11.5, the renewal period $T_{|\mathcal{V}|}$ is set to grow exponentially with the group size $|\mathcal{V}|$ as follows

$$T_{|\mathcal{V}|} = 2^{(|\mathcal{V}|-1)} \cdot \texttt{RenewalRate}$$

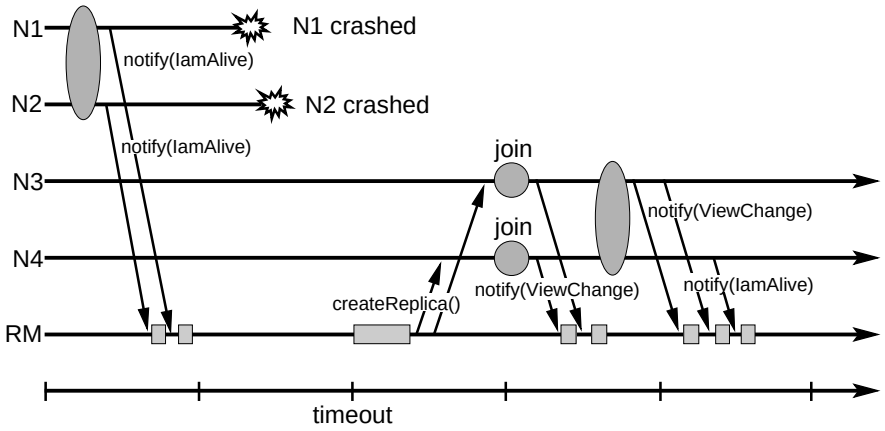


Figure 11.4: A simple group failure scenario. The timeout indicates that the expected renew event was not received, and hence ARM activates recovery.

where `RenewalRate` is obtained from the service configuration, e.g. Line 28 in Listing 10.2. This keeps the failure detection time short for small groups that are more likely to fail without notifying the RM, while reducing the number of renew events for larger groups that are less likely to experience a group failure. Hence, the overhead of this mechanism can be made insignificant compared to traditional failure detectors.

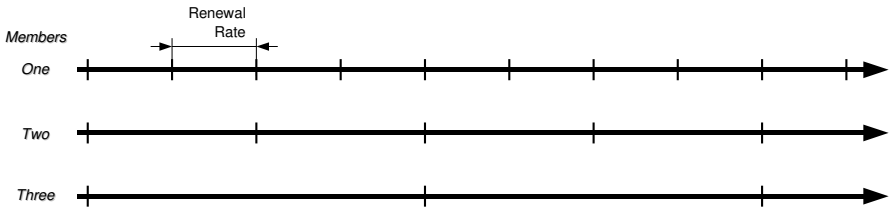


Figure 11.5: The renew rate grows with the group size.

Note that Jgroup/ARM do not provide support for reestablishing state in case of a group failure. Hence, recovering from a group failure is mainly useful to stateless services; alternatively the service may provide its own state persistence mechanism. Since the lease renew mechanism may not be useful to all services and does in fact induce some overhead, it can be (de)activated through the `GroupFailureSupport` property in the service configuration (see Line 27 in Listing 10.2). Note also, that

the client-side group proxy makes no attempt to handle group failures, and once they occur the proxy will notify the client through a GroupUnreachableException. Hence, it is left to the client application to handle this event, e.g. by retrying after a suitable delay, giving the service a chance to recover.

11.3.3 The Remove Policy

In addition to monitoring the group membership of its associated service, the supervision module also provides a controlling part, or more precisely a *remove policy*. Let $\mathcal{V}$ denote a view and $|\mathcal{V}|$ its size. If $|\mathcal{V}|$ exceeds the initial redundancy level R_{init} of the service, for a duration longer than a configurable time threshold (see `RemoveDelay` in Line 6 in Listing 10.2), then one excessive replica is requested to leave the group. If more than one replica needs to be removed, each remove is separated by the `RemoveDelay`. This remove policy is motivated by the desire to maintain a high redundancy level for a longer period of time, when recovering from a network partition that may become active again. A different remove policy could easily be implemented in which all needless replicas are remove after the initial `RemoveDelay`.

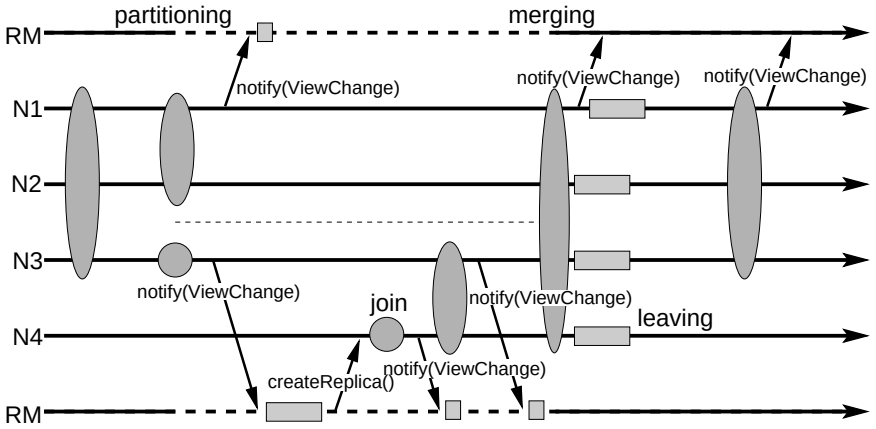


Figure 11.6: A sample network partition failure-recovery scenario. The partition separates nodes $\{N1, N2\}$ from $\{N3, N4\}$. The dashed part of the RM timelines indicate the duration in which the two replicas reside in separate partitions. After merging, the supervision module detects one excessive replica, and elects $N4$ to leave the group.

The choice of which replicas that should leave is made deterministically based on the view composition; in this way, the removal can be performed in a decentralized way, without involving the RM. This mechanism is illustrated in the last part of Figure 11.6, where the replica on node $N4$ decides to leave the group, consequently bringing the group back to a triplicated group.

The reason for the presence of excessive replicas is that during a partitioning, the RM may have installed additional replicas in one or more partitions to restore a minimal redundancy level, as shown in Figure 11.6. Once partitions merge, these replicas are in excess and no longer needed to satisfy the replication policy. Chapter 16 provides measurements of the time to restore R_{min} when exposed to a network partition and the time to remove needless replicas when merging.

As mentioned in Section 11.1, when installing a group using the management client two or more RM replicas may in rare circumstances operate in concurrent views, causing ARM to create the same set of service replicas in distinct subparts of the target environment. Also this problem is circumvented by the removal of the needless replicas.

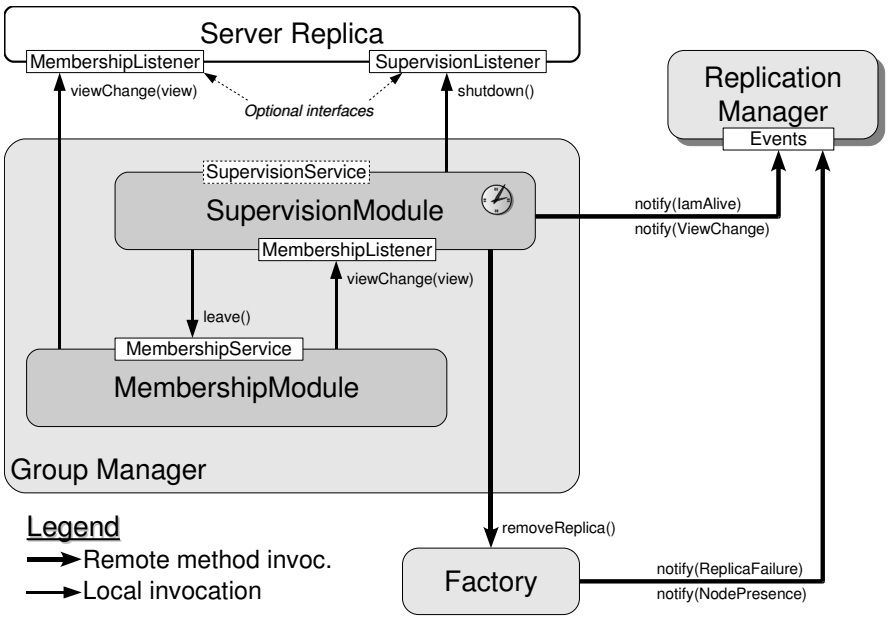


Figure 11.7: Partial set of the protocol modules required by ARM deployed replicas.

Figure 11.7 shows a slightly more detailed view of the SupervisionModule than previous figures. Once a replica has been selected for removal by the remove policy, the SupervisionModule invokes the leave() method on the MembershipModule. The server replica may optionally implement the SupervisionListener interface, and if so the shutdown() method is called by the SupervisionModule after the replica has left the group. The final task performed is removeReplica() on the Factory, which ultimately results in the destruction of the replica. The SupervisionService interface is shown as a dashed boxed indicating that it contains no methods, since the SupervisionModule is self-controlled based only on the input from the MembershipModule, and on the lease renewal timer associated with the replica level monitoring.

11.4 The Object Factory

The purpose of object factories is to facilitate installation and removal of service replicas on demand. To accomplish this, each node in the target environment must run a JVM hosting an object factory, as shown in Figure 11.1. In addition, the object factory is also able to respond to queries about which replicas are hosted on the node. The factory also provides means for the RM to keep track of available nodes.

The factory maintains a table of local replicas; this state need not be preserved between node failures since all replicas would have crashed as well. Thus, the factory can simply be restarted after a node repair and continue to support new replicas. Note that if only the factory JVM fails, a new factory instance will be unable to connect to the running replicas. However, those replicas will continue to provide service to clients independent of the factory, and if they fail, the RM will detect this and take appropriate recovery actions.

Object factories are not replicated and thus do not depend on any Jgroup or ARM services. This allows the RM to bootstrap itself onto nodes in the target environment using the same distribution mechanism used for deploying other service replicas. The RM may create new replicas and remove old ones by invoking the factory of a node. Replicas normally run in separate JVMs as shown in Figure 5.1, to avoid that a misbehaving replica causes the failure of other replicas within a common JVM.

During initialization, each factory looks for a running RM in the target environment by querying the DR. If RM is present, a NodePresence event is sent to make the RM aware of the newly available node. If the RM is not present when the factory is created, the registration of the new node is postponed until the RM is started. At

that point, all nodes in the target environment will be probed by the RM for running factory objects using the ping() method. Together, these two mechanisms enable the RM to become aware of all nodes that are capable of hosting replicas.

This probing mechanism is also used by ARM to determine if a node is available before selecting it to host a replica. In addition, the factory monitors the connection between the factory and the replica process, and sends a ReplicaFailure event to the RM if the replica process fails. This is primarily used by ARM to detect replica startup failures.

11.5 Failure Recovery

Failure recovery is managed by the RM, and consists of three parts; (i) determine the need for recovery, (ii) determine the nature of the failures, and (iii) the actual recovery action. The first is accomplished through a *reactive* mechanism based on service-specific timers, while the last two use the abstractions of the replication and distribution policies, respectively.

The RM uses a timer based *Service Monitor* (SM) to keep track of the installed replicas. When deploying a service, a new SM timer instance is associated with that service. If the scheduled expiration time of the SM timer is reached, the recovery algorithm is invoked. To prevent activating unnecessary recovery actions, the SM timer must be rescheduled or canceled before it expires. The SM timer expiration time is configured in the replication policy of each service, e.g. see Line 18 in Listing 10.2, and is a safety margin to avoid premature recovery activation. The ViewChange events reported by the SupervisionModule are used to determine if a SM timer should be rescheduled or canceled. If the received view $\mathcal{V}$ is such that $|\mathcal{V}| \geq R_{\text{min}}$, the SM timer is canceled, otherwise the SM is rescheduled to await additional view changes. Evidently, if the last view change $\mathcal{V}$ received when the timer expires, is such that $|\mathcal{V}| < R_{\text{min}}$, then the RM triggers recovery. Since each service has a separate SM timer, the RM is able to handle multiple concurrent failure activities in separate services, including failures affecting the RM itself.

When deploying a service, the RM will instantiate a service-specific replication policy. During its operation, the RM receives events and maintains state associated with each of the deployed services, including the redundancy level of services. This state is used by the replication policy to determine the need for recovery.

Upon expiration of the SM timer and detecting that the service needsRecovery(), the recovery algorithm is executed (see Algorithm 11.1 and Algorithm 11.2). The purpose of the recovery algorithm is to determine the nature of the current failure scenario affecting the given service, and to initiate recovery. Recovery is performed through three primitive abstractions: restartReplica(), relocateReplica() or groupFailure() as discussed in Section 10.1.2. The restart abstraction is used if a replica JVM that was supposed to be running on a node has failed, but the node's factory remains available. The relocation abstraction is used if the node is considered unavailable, i.e. that both the replica and the factory is not responding. Finally, the group failure recovery abstraction is only used if all service replicas have failed, and will typically reuse the relocation abstraction. The actual installation of replacement replicas is accomplished using the distribution policy.

Algorithm 11.1 The recovery algorithm (simplified)

```
1: recover(service)                                   {RECOVER THE GIVEN SERVICE}
2:   rpolicy ← service.getReplicationPolicy()
3:   rpolicy.prepareRecovery()
4:   recovered ← rpolicy.handleFailure()
5:   if recovered
6:     service.rescheduleMonitor()                    {Reschedule the service monitor}
7:   else
8:     service.notRecovered()                         {Failed to recover; log problem}
```

11.6 Replicating the Replication Manager

The RM is a centralized, yet critical component in our framework. If it were to crash, future replica failures would not be recovered from, thereby severely damaging the dependability characteristics of the system. Also, it would prevent the installation of new services for the duration of its downtime. Therefore, the RM must be replicated for fault tolerance, and it must be able to recover from failures affecting the RM itself, including network partition failures. Careful consideration is required when replicating the RM; one needs to consider the consistency between RM replicas in face of non-deterministic input as well as the merging of states after a network partition scenario. If the RM were to crash, services deployed through ARM will continue to provide service to its clients independent of the RM. Note that after an RM failure, replicas could potentially reconnect to the RM and regain recovery support. However, this would require an additional mechanism to reestablish the state of the RM.

Algorithm 11.2 The KeepMinimalInPartition replication policy (partial)

```
1:  KeepMinimalInPartition(theService, distPolicy)
2:    service ← theService                          {The service associated with this policy}
3:    assigned ← distPolicy.getAssignedNodes(service)   {The set of assigned nodes}
4:    missing ← ∅                                   {The set of missing members}
5:    V ← ∅                                         {The current view}
6:  notify(view)                                    {VIEW UPDATE}
7:    V ← view
8:  handleFailure()                                 {GENERIC FAILURE HANDLER}
9:    recov ← true
10:   if |V| = 0
11:     recov ← groupFailure(missing)               {All members have failed; group recovery}
12:   else
13:     foreach node ∈ missing
14:       if node.isSuspected()
15:         recov ← recov ∧ relocateReplica(node)   {The node has failed; relocate}
16:       else
17:         runningServices ← node.getServices()    {Get the services running on node}
18:         if service ∉ runningServices
19:           recov ← recov ∧ restartReplica(node)  {Service has failed on node; restart}
20:   return recov
21: needsRecovery()                                 {CHECK THE NEED FOR RECOVERY}
22:   return |V| < R_min                            {Return true if recovery needed}
23: prepareRecovery()                               {PREPARE FOR RECOVERY}
24:   missing ← ∅
25:   if assigned ≠ V
26:     missing ← assigned − (assigned ∩ V)
27: relocateReplica(node)                           {RELOCATE REPLICA TO NEW NODE}
28:   newNode ← distPolicy.reassignReplica(service, node)
29:   recov ← newNode.createReplica(service)
30:   return recov                                  {Return true if replica created successfully}
```

To make appropriate (recovery) decisions, the RM relies on non-deterministic inputs, such as the SM timers. These inputs are affected by events received by the RM as shown in Figure 11.2. Hence, to prevent RM replicas from making inconsistent decisions, only the group *leader* is allowed to generate output. The leadercast semantic is used for the methods that perform non-deterministic computations that always update the state, e.g. createGroup(), while multicast semantic is used for the notify() method. Stronger invocation semantics are not required since invocations related to different groups are commutative. Although notify() is a multicast method, only the RM leader replica is allowed to perform the non-deterministic part of the processing and inform the *follower* replicas if necessary. For example, only the leader replica

will actually perform a recovery action, while the followers are only informed about the new location of the replica.

As the replication protocols are implemented on top of the group membership module, leader election can be achieved without additional communication, simply by using the total ordering of members defined in the current view. If the current leader fails, a new view will be installed excluding the current leader, and in effect a follower replica will become the new leader of the group and will be able to resume processing.

Since the RM is designed to tolerate network partition failures, it may in *rare* circumstances cause temporary inconsistencies due to EGMI events being handled in multiple concurrent views. However, in most cases, inconsistencies will not occur since each replica of the RM is only "connected" to replicas within its own partition. That is, most events (e.g. the location of replicas determined from view change events) received by the RM replicas reflect the current network partitioning. Hence, a potential inconsistency will be recovered from as soon as additional events cancel them out. If an inconsistency were to persist long enough to cause the RM to activate an unwarranted recovery action, the supervision module would eventually detect this and remove the excessive replicas. Hence, the application semantics of the RM described above enables it to tolerate partition failures; a feature that by far outweighs the sacrifice of slightly weaker consistency. The impact of weaker consistency can only result in higher redundancy levels.

When merging from a network partition scenario (see Figure 11.6), the RM invokes a reconciliation protocol using the state merging service (see Section 3.3.3), to merge the locations of service replicas. This is feasible since the location of service replicas in each merging partition will, after the merging, be visible to all RM replicas in the merged partition. In addition, the reconciliation algorithm also restarts the SM timers of the involved services, since the RM leader replica of the merged partition might have received information about new services during reconciliation. The latter is primarily a safety measure to prevent premature recovery actions.

The RM relies on the dependable registry (DR) service to store its object group reference, enabling RM clients such as the supervision module, factory and management client to query the DR to obtain the group reference of the RM. Due to this dependency, ARM has been configured to co-locate RM and DR replicas in the same JVM (see Figure 5.1). This excludes the possibility that partitions separate RM and DR replicas, which could potentially prevent the system from making progress.

As mentioned previously, the RM exploits its own embedded recovery mechanism to handle self-recovery in case of RM replica failures. The exception being that the RM cannot tolerate a group failure, since it makes little sense in sending IamAlive events to itself.

11.7 Summary

The ARM framework presented in this chapter is a fault treatment system that is able to reconfigure the system to reduce the consequences of failures. The centralized RM component is responsible for making decisions concerning the activation of recovery, i.e. to install replacement replicas to restore the desired level of redundancy. Furthermore, to avoid too high redundancy levels, replicas may be removed as well. The remove policy is implemented in a decentralized manner.

The performance cost of group level monitoring and the remove policy is very low, since both techniques exploits synergies with the Jgroup PGMS (see Section 3.3.1). Moveover, the cost of the replica level monitoring technique can also be made quite low since it is merely a supplementary technique to group level monitoring.

The small cost of the techniques provided by ARM results in a negligible overhead during normal operation. In the recovery phase, ARM responds rapidly by installing replacement replicas as demonstrated by the measurements in Part IV.

Chapter 12

Online Upgrade Management

Most distributed software systems evolve during their lifetime. The spectrum of software change is wide, and ranges from program corrections and performance improvements to complex changes of the overall functionality, configuration and structure of the system. Such changes may be necessary to adapt the system to new user requirements. In a conventional approach to system maintenance, the system is taken offline during maintenance, and often the necessary changes are manually applied to the system. This approach is unsuitable for distributed systems with strict availability requirements.

To effectively handle maintenance changes, support for online upgrade management must be provided. However, managing the changes at runtime in highly available distributed systems is especially challenging, as upgrading a running system should not deteriorate its availability characteristics. Moreover, when used in conjunction with a fault tolerant system based on replication, online upgrade techniques can be implemented to improve the service availability characteristics, by eliminating (or reducing) the downtime during maintenance activity.

The purpose of this chapter is to demonstrate that an algorithm for online upgrading can easily be implemented in the context of the Jgroup/ARM framework. This chapter is based on joint work with Marcin Solarski [115]. The upgrade algorithm presented briefly in this chapter is due to Solarski, whereas the main contribution is a simple architecture for online upgrade management built on Jgroup/ARM. For further details about the upgrade algorithm, see [115, 114].

Chapter outline: Section 12.1 introduces the concept of software upgrading from several viewpoints. In Section 12.2 the underlying system model is presented along with

our assumptions for the upgrade algorithm. In Section 12.3 we describe the architecture of the upgrade system based on Jgroup/ARM, and in Section 12.4 we briefly present the upgrade algorithm. In Section 12.5 an alternative upgrade approach is proposed which is expected to be significantly more efficient than the original algorithm in [115]. Finally, Section 12.6 provides a few closing remarks.

12.1 Introduction

Traditional techniques for increasing system availability have been based on masking failures [108]. The general idea is to introduce redundancy into the system by replicating critical system components, eliminating the effects of transient hardware and software failures. However, replication cannot prevent system failures due to software design faults whose contribution to system unavailability accrues rapidly with the increasing complexity of software systems. By upgrading the software, the number of design faults in a system can be reduced. Online upgrade is a technique that allows the introduction of necessary changes into the system, so that the system remains operational even while being upgraded. This is possible by upgrading only a subset of the replicas, while the other subset remains operational and serving clients. Thus, system availability does not decline as a result of the system upgrade.

Online upgrading of software entities is a research field of its own and a huge body of research already exists in the literature [53, 67, 110, 127, 114]. The aim of this chapter is not to compete with previous works, merely to demonstrate that the Jgroup/ARM platform is easily enhanced to support online upgrades. The unit of upgrade considered in previous work ranges from a single operation to functions, programs and even distributed subsystems. Also most previous works are mainly focused on upgrading non-replicated software entities. In our approach, the unit of upgrade is a replicated object and focus is on the dependability characteristics of the upgrade process. The Eternal Evolution manager [127] supports live upgrades of actively replicated objects using an approach similar to ours. The target of an upgrade may comprise a set of CORBA objects, both clients and servers. The upgrade proceeds by replacing single replicas in two phases, while the object group as a whole remains operational for the duration of the upgrade. The first phase involves an intermediate version, used to allow additional flexibility in the permitted changes. This, in contrast to our one-phase upgrade algorithm, is achieved through additional complexity.

In [114] online upgrade management is considered from three different viewpoints as follows:

- **System Evolution** Online upgrades can be considered as a method to handle change management in evolving software systems at runtime.
- **Software Deployment** Online upgrades can be viewed as a special case of service deployment, in that previously deployed software is replaced by a new version.
- **High Availability** Online upgrades can also be viewed as a means to improve the service availability characteristics, by reducing or eliminating the required downtime during maintenance activity.

The primary concern in this chapter is availability, however, the software deployment aspect is also covered through the ARM framework presented in Chapter 11.

The objective of replicating system components is to reduce the number and duration of system downtimes; the evaluation in Chapter 15 demonstrates the importance of short recovery times in this respect. However, unless online upgrades are supported, system downtimes due to (manual) upgrades will completely dominate the system's availability characteristics, independent of replicating system components.

12.2 System Model and Upgrade Assumptions

The upgrade management framework inherits the system model of Jgroup/ARM, as described in Section 3.1, except that network partition failures are not handled explicitly by the upgrade algorithm presented later. The algorithm assumes that server replicas fail only by crashing, and once crashed it does not recover. However, a replica that is considered to have crashed may be replaced by a new instance of the replica. A new replica instance may be the same or a new version of the replica.

We consider only upgrading the software of server replicas and not the clients. This assumption places certain restrictions on what can be achieved with respect to compatibility between client and server objects. Let $\hat{v}$ denote the current version of a replica to be upgraded to version $\hat{v} + 1$. Thus, in order to substitute a replica of version $\hat{v}$ with a replica of version $\hat{v} + 1$, the upgrade algorithm makes the following assumptions:

- *Upgrade atomicity with respect to other upgrades of the server.* Server upgrades are atomic with respect to each other, i.e. two upgrade processes cannot interleave. Furthermore, a replica cannot process client requests while being upgraded.
- *Input conformance.* Replica version $\hat{v}$ is replaceable with version $\hat{v}+1$. In terms of input, the input accepted by version $\hat{v}+1$ is the same as the input acceptable to version $\hat{v}$ of the replica, possibly augmented with new inputs. In terms of interfaces, it is assumed that version $\hat{v}+1$ offers a compatible interface to that of version $\hat{v}$, possibly augmented with new functionality.
- *State mapping and output conformance.* There exist a mapping from the state of version $\hat{v}$ to the state of version $\hat{v}+1$ of the replica, such that version $\hat{v}+1$ produces the same output as version $\hat{v}$, given input acceptable to version $\hat{v}$.
- *Upgrade atomicity with respect to client upgrades.* Clients that generate input, acceptable to version $\hat{v}+1$, but not acceptable to version $\hat{v}$, do so only after the upgrade algorithm terminates.

Furthermore, code downloading mechanisms are not dealt with in this work, and instead it is assume that code for the new software version $\hat{v}+1$ has been deployed to all the system nodes and can be instantiated.

12.3 A Simple Architecture for Online Upgrades

A simple architecture for supporting online upgrades of replicated services deployed using the Jgroup/ARM framework is presented. The architecture extends on the ARM framework as shown in Figure 12.1.

Let $S_A(*)$ denote the server replicas to be upgraded. The *upgrade manager* (UM) is responsible for mediating upgrade requests to the UpgradeModule of the respective replicas in the group. The UM implements the UManagement interface which contains two methods used by the management client to perform upgrades/downgrades. The upgradeGroup() method is parameterized with version information of the application to be upgraded. The upgrade manager is co-located with the replication manager; in fact it simply extends the replication manager with the UManagement interface.

The upgrade algorithm described in Section 12.4 is implemented by the UpgradeModule. The main task of the UpgradeModule is to drive the upgrade process triggered through the upgradeRequest() method. It is also responsible for interacting with the local Factory to create a replica of the new version.

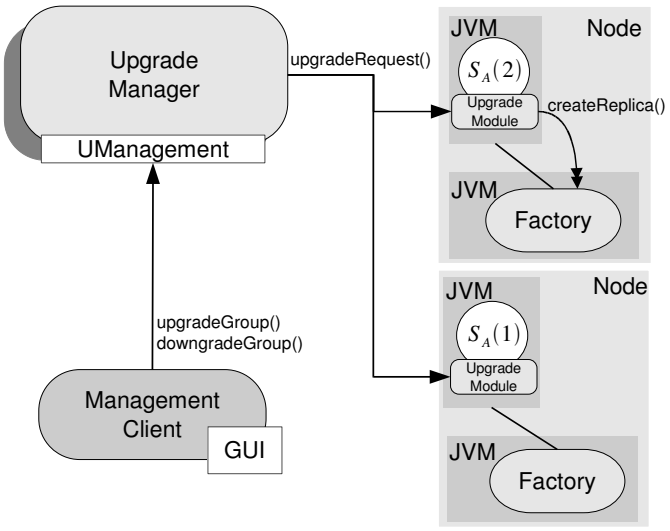


Figure 12.1: The Jgroup/ARM online upgrade architecture.

12.3.1 The Upgrade Module

Figure 12.2 shows the protocol modules and interfaces used by upgradable replicas. In response to an upgradeRequest(), the UpgradeModule determines if the local replica should be upgraded, and if so the local factory is invoked to create a replica with the new version. Once the new replica has been installed and joined the group, the old replica is requested to leave the group. Recall that a view is an ordered set of member identifiers. Hence, to distinguish the old and new replicas in the same view, the member identifier is augmented with a version number.

The choice of which replica to be upgraded in each iteration of the upgrade algorithm is made by the UpgradeModule, based on the relative position of the replica in the current view. Hence, the UpgradeModule needs to receive viewChange() events from the MembershipModule. These tasks are seamlessly handled by the UpgradeModule.

The application replica may optionally implement the UpgradeListener interface, i.e. the upgraded() method. The upgraded() method is invoked by the UpgradeModule to notify the replica that a new version *has been* installed and that this replica has left the group; the replica may then gracefully shutdown. Note that the replica is not required to implement the UpgradeListener interface; the UpgradeModule will invoke the removeReplica() method on the factory (after having returned from the upgraded() method) to ensure that the old version is removed.

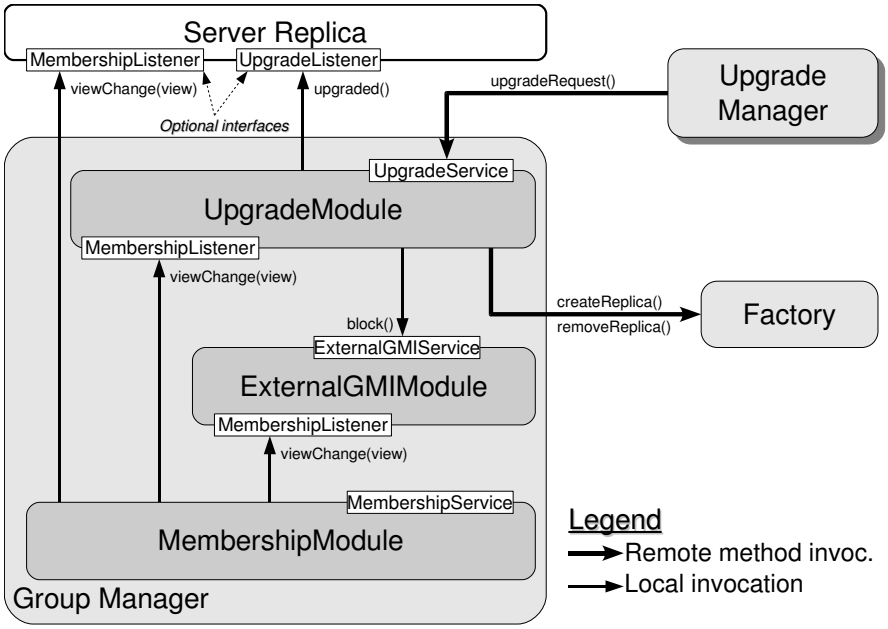


Figure 12.2: Protocol modules required by upgradable replicas.

Assuming that the application is stateful, the new version of the replica must ensure that its state is synchronized with the remaining members of the group, before it can start processing client requests. State synchronization is not handled by the UpgradeModule, and instead this task is a concerted effort between the StateMergeModule (see Section 3.3.3) and the application. That is, the new application version needs to provide a translation method between the old and new state representation. This implies that the putState() method of the new application version must be able to handle state retrieved from both the new and old version.

Finally, to prevent client requests from being processed by the replica during an upgrade, the UpgradeModule interacts with the ExternalGMIModule, as indicated by the block() method. This is required to prevent returning potentially inconsistent results to clients while being upgraded.

12.4 The Upgrade Algorithm

In this section, we briefly present a software upgrade algorithm whose purpose is to replace the code of a running replicated service with a new version of the software.

The algorithm is designed to avoid single points of failure and it is implementable given the assumptions in Section 12.2. Additional details about the algorithm can be found in [115].

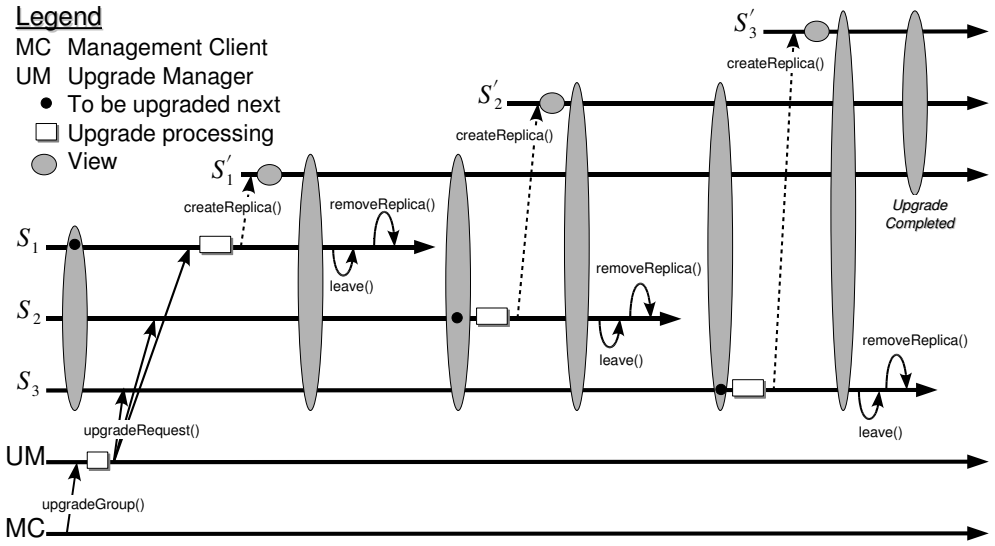


Figure 12.3: Upgrade interactions for a triplicated server group.

Prior to upgrading a particular application, it should first be deployed through the ARM framework (the replication manager). Figure 12.3 illustrates the interactions of an upgrade process. The choice to initiate an upgrade is made by the system operator through the management client (MC). The management client allows an upgrade to take place by invoking the upgradeGroup() method on the upgrade manager, which in turn leads to a upgradeRequest() multicast invocation directed towards the UpgradeModule of each replica in the group to be upgraded.

Next, the UpgradeModule of S_1 is to be upgraded first, consequently it invokes createReplica() on the local factory. The new version of the replica is named S_1' and joins the group, which installs a new view containing four members: $\{S_1, S_1', S_2, S_3\}$. When the UpgradeModule of S_1 detects this view, it requests to leave the group and commits suicide, consequently causing another view installation with three members again: $\{S_1', S_2, S_3\}$. The upgrade algorithm proceeds until all replicas have been replaced with the new version, eventually returning to a stable condition with $\{S_1', S_2', S_3'\}$.

12.4.1 Summary

Further details about the upgrade algorithm presented above is provided in [115]. This section summarizes the main properties of the upgrade algorithm:

- The algorithm requires that there be a minimum redundancy level $R_{\text{min}} > 1$, before a replica is replaced. Furthermore, if a replica cannot be upgraded it will continue to provide service using the old version. Thus, continuous availability is provided as there are replicas capable of processing client requests at any moment during the upgrade process.
- Replica state consistency is maintained through the state transfer mechanism provided by the StateMergeModule. State transfers are invoked for each upgraded replica. It is assumed that state transfer can be achieved across different versions of the replica, as stated in Section 12.2.
- The algorithm is fault-tolerant in that the algorithm coordination is decentralized and it tolerates replica crashes. As there is no single entity that controls the progress of the algorithm, the upgrade continues even in presence of crashes of the replicas being upgraded. The recovery mechanism provided by the ARM framework allows recovery from replica crashes by instantiating a new copy of the replica.
- At any time during the upgrade, only one additional replica is added to the group, thus keeping the number of replicas in the system to a minimum.

Note that the upgrade algorithm by itself does not guarantee maintaining the redundancy level. To maintain a given redundancy level for the group, also outside the upgrade phase, the ARM framework is used.

The implementation of the algorithm presented above has been evaluated experimentally by Solarski in the context of this dissertation [114]. His experiments were conducted with varying redundancy level and work load placed on the servers being upgraded. As expected the invocation-response latency increased during the upgrade phase; the increase varied significantly for the different work loads and redundancy levels. The average increase for a simple anycast method was 25%, whereas the increase for a multicast method was as high as 66%. The total upgrade time also varied significantly with the redundancy levels; for the anycast method the upgrade time ranges from 13 seconds for a two-way replicated service up to 22 seconds for a four-way replicated service. Solarski's experiments do not consider crash failures during the upgrade phase, since the ARM framework was not fully developed at the time.

Note that the client-side view refresh approach discussed in Chapter 9 is also useful when upgrading servers, as new versions of replicas will use a different local communication endpoint (port allocation), and thereby rendering the client-side membership information invalid. Also, to avoid the problem shown in Figure 9.3, the client-side proxy must support a mechanism to obtain a fresh copy from the DR after an upgrade.

12.5 An Alternative Upgrade Approach

The upgrade algorithm presented above exploits the view agreement protocol and the state transfer of the underlying group communication system (GCS) to ensure consistent behavior of the upgraded replicas. However, in doing so the duration of the upgrade depends heavily on the redundancy level of the service, and also clients invoking the service experience periods of blocking while the GCS is busy transferring state or running the view agreement protocol. For instance, the upgrade algorithm above requires $2R$ runs of the view agreement protocol as shown in Figure 12.3, where R denotes the redundancy level of the group.

In this section, an alternative upgrade approach is proposed in which the upgrade is performed locally only, and as such avoids lengthy runs of the view agreement protocol and remote state transfers.

Figure 12.4 shows interactions required for the local upgrade approach. Also the proposed technique upgrades the replicas one-by-one until they have all been upgraded. During the upgrade of a replica, the new version is created in the same JVM as the current replica instead of a separate JVM. Before activating the new replica, the state of the old replica must be transfered to the new replica and possibly translated into a new state representation. While being upgraded, the replica must block invocations; however, the other members of the group are still able to process invocations. After initializing the state of the new replica, invocations are redirected to the new replica instead. The old replica can then be garbage collected by the JVM.

Note that this approach is made possible through a combination of the dynamic protocol composition architecture (see Chapter 6) and the revised EGMI architecture (see Chapter 7). That is, these architectures allows us to exercise greater control over the local server references associated with the various protocol modules, including the method dispatching mechanism of the ExternalGMIModule. Another advantage of the local upgrade approach is that the DR and the client-side proxy does not need

to be updated since the server-side proxy remains the same across upgrades, i.e. the communication endpoint can be reused for the upgraded replica.

Further study of the proposed algorithm is needed to reveal its full potential and if there are problems with respect to fault tolerance and so on.

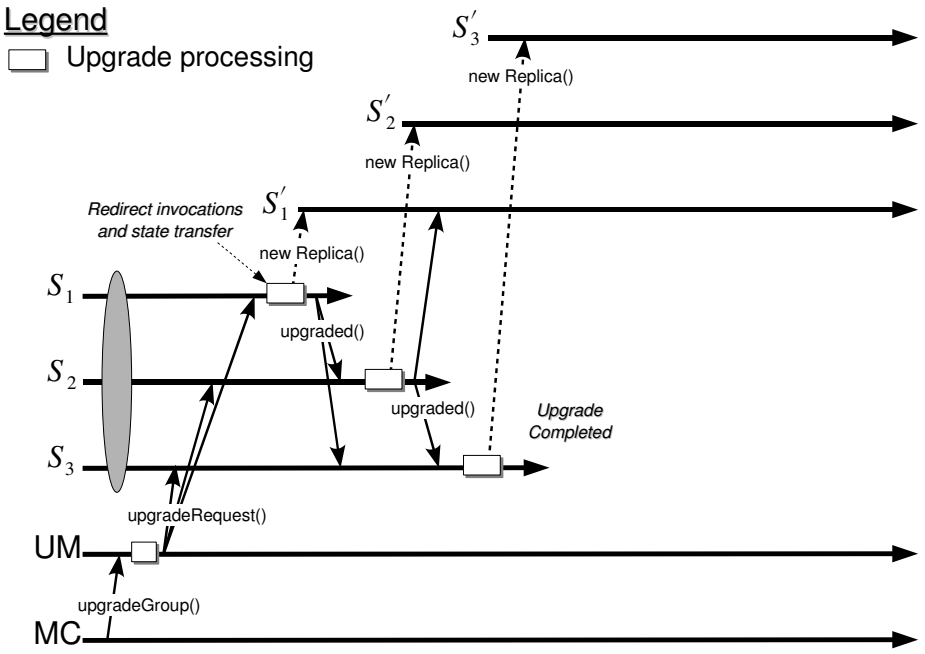


Figure 12.4: Upgrade interactions for the local upgrade approach.

12.6 Closing Remarks

The objective in this chapter has been to demonstrate the feasibility of implementing an algorithm for online upgrading of software components within the Jgroup/ARM framework. A simple architecture to support this objective has been developed. Experience with this implementation has shown that closer integration between upgrade and recovery policies is needed, or at least an improved awareness of both problem domains. Otherwise, conflicting policies are easily introduced. As an example, consider a fallback policy where a new version is replaced by the old version if a group failure occurs; such a policy needs the cooperation from both upgrade and recovery management components.

Part IV

Experimental Evaluation

Chapter 13

Toolbox for Experimental Evaluation

Performing experimental evaluation of fault tolerant distributed systems is a complex and tedious task, and automating as much as possible of the execution and evaluation of experiments is often necessary to test a broad spectrum of possible executions of the system to obtain good coverage. The confidence of the results obtained from an experimental evaluation depends on the degree of control over the environment in which experiments are being executed. Typically, an uncontrolled environment is exposed to numerous sources of external influence that can affect the obtained results. Automated and repeated executions can be used to reduce the impact of such influences.

In this chapter, a framework for experimental validation and performance evaluation of fault management in a fault tolerant distributed system is presented. The framework provides a facility to execute experiments in a configured target system. It is based on injecting faults or other events needed to test the fault handling capability of the system. Relevant events are logged and collected for post-processing and analysis, e.g. to construct a single global timeline of events occurring at different nodes in the target system. This timeline of events can then be used to validate the behavior a system, and to evaluate its performance.

This chapter is organized as follows: Section 13.1 introduces the concept of fault injection and discuss related work, followed by an architectural overview of the experiment framework in Section 13.2. In Section 13.3 we briefly describe the experiment scripting organization used by the framework. Section 13.4 explains the two fault

injectors used in the experiments and gives a brief inaccuracy analysis. Section 13.5 explains the general organization of the analysis modules, and finally in Section 13.6 the impact on the tested system and the accuracy of the instrumentation is discussed.

13.1 Introduction

Testing the validity of fault tolerant distributed systems and measuring the performance impact of faults is a challenging task. A common technique is to apply *fault injection* (see for instance [4, 5, 52]) as a means to accelerate the occurrences of faults in the system. The main purpose of fault injection is to evaluate and debug the error detection and recovery mechanisms of the distributed systems.

Numerous systems [4, 28, 35, 119] have been developed to provide generic fault injection tools aimed at testing the fault handling capability of systems. The most relevant ones are discussed briefly below. Loki [28] is a global state-driven fault injector for distributed systems. Faults are injected based on a partial view of the global state of a system, i.e. faults injected on one node of the system can depend on the state of other nodes. Loki has been used to inject correlated network partitions to evaluate the robustness of the Coda distributed filesystem [71]. Orchestra [35] is a script-driven probing and fault injection tool designed to test distributed protocols. It is based on inserting a fault injection protocol layer below the target protocol that will filter and manipulate messages exchanged between protocol participants. Since a separate layer is used, the source code of the tested application does not need to be modified. NFTAPE [119] is a software infrastructure for assessing the dependability attributes of networked configurations. The main feature of NFTAPE is extensibility, and is so in two ways: (i) a suite of tools to support specifying injection scenarios, and (ii) a library of injection strategies and a light-weight API to customize injection strategies or develop new ones. Each machine in the target system is associated with a process manager which communicates with a centralized controller. The centralized controller injects faults according to a specified fault scenario by sending commands to the process managers.

It is unclear if any of these frameworks are available for others to use. But more importantly, none of the frameworks match the needs of our experimental evaluation. Since they are not specifically designed for testing distributed Java applications, significant effort would have been required to adapt these systems.

So instead we have designed a simple and modular experiment framework specifically for testing Jgroup/ARM. A significant portion of the Jgroup/ARM APIs is

reused by the framework to ensure consistency between the various configurable parameters of the target system. It provides a facility for execution of experiments in a configured target system, and enables the injection of faults to emulate realistic failure scenarios.

Terminology To experimentally evaluate a system, one or more *studies* may be defined, e.g. a crash failure study (see Chapter 15) or a network instability tolerance study (see Chapter 16). For each study, one or more *configurations* are defined; a configuration typically specify the target system and deployment parameters such as the number of replicas for each service. However, in the following only a single configuration per study is considered. To obtain statistically significant measures, several runs of each study are performed. Each of these runs is called an *experiment*. Each study (and configuration) is evaluated separately. After each experiment, the system is reset to its original configuration before beginning the next experiment. The above terminology is partially borrowed from [28].

During an experiment, events are logged. The set events to be logged are defined *a priori*, and the code is instrumented with logging code. After the completion of an experiment the log files are collected for analysis. Experiment analysis is specific to each study and typically involves the construction of a single global timeline of events occurring at the different nodes in the target system. This global timeline of events can then be used to validate the behavior of the system, and to evaluate its performance. For instance, a predefined state machine for the system behavior may be used to validate the behavior of the system by projecting the event trace onto the state machine.

13.2 Architectural Overview

The experiment framework is designed to perform repeated experiments of a study to obtain statistically significant measures. Figure 13.1 shows a generalized view of the components of the experiment framework. The main component is the *experiment executor*. Its purpose is to execute scripts defining a study. In each experiment numerous tasks are executed, e.g.:

1. Reset and initialize the nodes in the target system
2. Bootstrap the object factories onto the nodes in the target system

3. Bootstrap the ARM infrastructure (RM replicas)
4. Deploy the replicas
5. Inject faults
6. Shutdown the experiment
7. Collect log files from the target system nodes

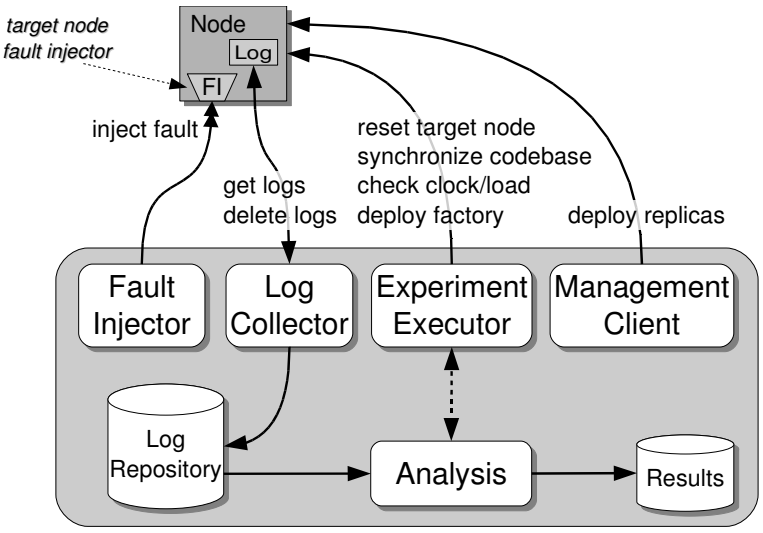


Figure 13.1: Experiment framework architecture.

Each component in the architecture is defined by a set of tasks that it performs. Tasks are building blocks for constructing study scripts, and each script is comprised of a set of common tasks and a set of specialized tasks. For example, the specialized fault injector and analysis tasks used for the two experiments in Chapter 15 and Chapter 16 are completely different. The experiment executor interacts with all the other components to activate tasks according to the study script.

The target system nodes must host a fault injector through which faults can be injected. Depending on the type of faults being injected the fault injector code may have to be incorporated within the running system on the target node.

The events of interest must be logged for use in the analysis phase, and typically requires additions to the code. The log files are collected from each node in the target system and stored in a repository for post-processing.

During a study, the CPU/IO activity of the nodes in the target system is checked before and after each experiment. Experiments whose load exceeds some configurable threshold may then be marked for further analysis. This is particularly useful to detect artifacts due to external influence when the study is performed in an uncontrolled environment.

The analysis component is organized in two separate modules; one module to process each experiment individually and another module to process all experiments in the study collectively and produce statistics. Typically, the latter module will use the former to obtain measurements from each individual experiment. Note that each experiment may be analyzed after its completion, and the results of the analysis can be used by the experiment executor to make decisions; hence the dashed arrow between the experiment executor and the analysis component in Figure 13.1. The analysis component is discussed further in Section 13.5.

13.3 Experiment Scripting

The initial version of the experiment framework used XML based scripts to specify the experiment tasks to be executed; the experiment tasks themselves were implemented in Java. The study discussed in Chapter 15 was performed using the XML based framework. However, lack of support for control flow mechanisms in XML made it difficult and unnatural to write advanced study scripts. Therefore the experiment framework was ported [133] to the Groovy [47] scripting language, making it much easier to prepare complex study scripts. Groovy allows close integration with the Java language, thus enabling reuse of several Jgroup/ARM APIs.

Study scripts are typically organized in four phases, each executing various tasks:

1. **Initialization** The static configuration of the study is initialized. Dynamically adjustable parameters of the study are embedded within the experiment tasks below.
2. **Pre-study tasks** Tasks performed only one time before the actual study begins. This typically involves the creation of a log repository on the experiment machine and synchronizing the codebase of all the nodes in the target system.
3. **Experiment tasks** The main tasks needed to perform the study; these are repeated for each experiment. These tasks typically include: deploying the factories and replicas on the target system nodes, and injecting faults into the

nodes in the target system. After the execution of an experiment, the logs are collected from the nodes in the target system and the nodes are reset, e.g. by killing any remaining experiment processes and deleting log files.

4. **Post-study tasks** Tasks performed only one time after the completion of the study. For example, to remove log files from the target system nodes.

13.4 Code Instrumentation

Instrumenting code for our experiments is done by inserting logging statements and other code directly inside the actual source code of the system under study.

13.4.1 The Logging Facility

To simplify the logging of various system and failure events a logging facility is provided. A particular *event* is recorded by logging calls inserted at appropriate locations in the source code. Each recorded event includes:

- The time of the event; recorded using the local processor clock.
- Machine name on which the event was recorded.
- Event type and a brief description.

The recorded events are Java objects and support is provided for ordering the events into a single global timeline independent of the node on which the events occurred. Such ordering requires that the processor clocks of all the nodes in the target system are synchronized using NTP [83]. The granularity of the clock is one millisecond. Nanosecond granularity is also possible for computing the relative time between events occurring on the same node. The precision obtained using NTP is in the range 1-5 ms, according to the offset values obtained from the `ntpdate` command. This level of accuracy makes it very unlikely that events recorded on different nodes are ordered incorrectly in the global trace. Note that the clock offset values of each node are checked before and after each experiment to detect deviations above some threshold. Experiments with too large a clock deviation may be marked and excluded from further consideration.

Note that in the measurements presented in Chapter 15 events may also be ordered incorrectly for other reasons. These inaccuracies were solved by other means as discussed in Section 15.2.1.

The event class used by the logging facility may be subclassed to include event-specific details. For instance the *view event* subclass includes the view object generated by the PGMS (see Section 3.3.1). Event classes may also provide methods that can be used in the analysis phase to extract various properties from the event, for instance to check if a view event represents a fully replicated view.

To reduce the processing overhead of event logging, events are first stored in memory and periodically flushed to disk. However, to avoid loss of events in response to fault injections, the flush mechanism can also be triggered immediately before a fault injection.

13.4.2 Fault Injectors

The experiment framework currently supports two distinct randomized fault injectors; both implemented by means of code instrumentation:

- Crash failure injection (see Chapter 15)
- Reachability change injections (see Chapter 16)

13.4.2.1 The Crash Failure Emulator

The crash failure semantic is discussed in Section 2.1.1. To support crash failure emulation, the factory has been instrumented with a shutdown() method. The shutdown() method simply sends a terminate signal to the replicas associated with the factory, forcing each replica to halt its execution. Figure 13.2 illustrates the crash failure injector.

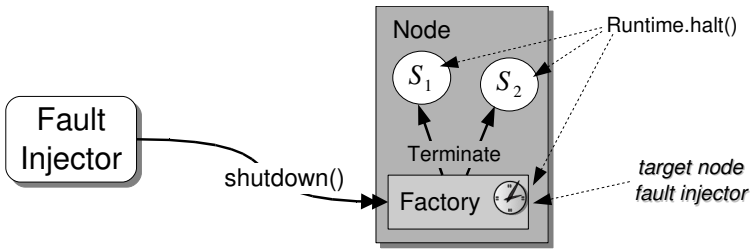


Figure 13.2: The crash failure injector.

When injecting multiple crash failures in a single experiment, all injections are sent to their respective nodes at the start of the experiment. A timer mechanism is then used

Table 13.1: Statistics for crash injection time (milliseconds). The injection time is measured from the crash activation time in the factory to immediately before halting the replica JVMs. The results were obtained from 100 crash failure experiments.

Mean	StdDev	Max	Min
13.833	4.772	32	8

to trigger the injections at the specified activation times. This way the communication step has a very low impact on the injection time accuracy.

Inaccuracy Injections are performed using the fastest possible way to stop a Java virtual machine from within itself, namely using the Runtime.halt() method. This means that no shutdown hooks or finalizers are executed during the shutdown sequence, as would be the case if we used the System.exit() method.

However, measuring the accuracy of crash injections is difficult since it is not easy to accurately detect the time when the process ceases to exist. Hence the measurements presented in Table 13.1 only accounts for the time taken from activating a crash failure at the specified time in the factory until immediately before the halt method is invoked in the replica JVMs. The results indicate that crash fault injections are quite fast, and hence do not contribute to any significant inaccuracy in the measurements in Chapter 15.

13.4.2.2 The Network Partition Emulator

At any given time, the connectivity state of the target environment is called the current *reachability pattern*. The reachability pattern may be *connected* (all nodes are in the same partition), or *partitioned* (failures render communication between subsets of nodes impossible). The reachability pattern may change over time, with partitions forming and merging as illustrated in Figure 13.3. The letters x, y and z each denote a different site in the target environment. Injections causing a transition from one reachability pattern to another is called a *reachability change*. A reachability change is due to either a *partition* or a *merge* event. Note that in the study in Chapter 16 the injected reachability patterns are assumed to be *symmetric*. However, *asymmetric* reachability patterns are easily supported by the partition emulator.

Injecting and measuring real network partitions in a wide area network is difficult for a number of reasons: (i) lack of physical access and permissions to disconnect cables

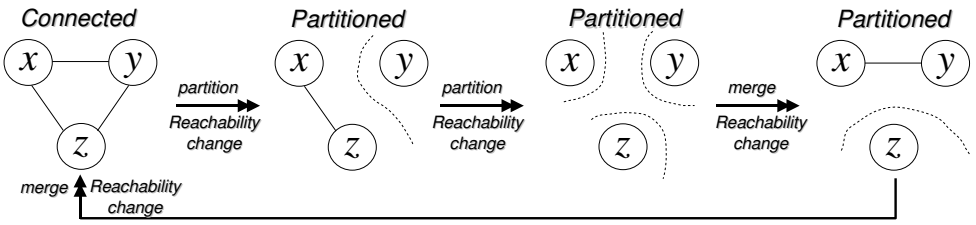


Figure 13.3: A sequence of reachability patterns.

from switches/routers, (ii) it is difficult to measure the exact time of disconnection, and (iii) performing a large number of disconnections would be very time consuming. For these practical reasons, network partition scenarios are instead emulated. To accomplish this, Jgroup/ARM has been instrumented with code to emulate network partition scenarios.

The partition emulator allows us to remotely configure and inject the reachability changes to be seen by the various nodes in the target system. Each node in the target system has a local partition emulator module through which injections are managed by the experiment executor (see Figure 13.1). The node local partition emulator is implemented by intercepting and discarding packets according to the configured reachability pattern. However, to avoid complicated changes to Jgroup/ARM, packet discarding must be done at the receiver, rather than the sender side. Hence, packets from ”disconnected“ nodes are also received and do require some minor processing. In our experiments in Chapter 16 this processing is negligible however, since there are no clients generating traffic.

The injection of a new reachability pattern is organized in a *setup phase* and a *commit phase*. The former configures the reachability change to be injected, while the latter activates it. The setup phase also serves to establish TCP connections to be reused in the commit phase. The setup phase must be performed before the injection time. Figure 13.4 illustrates the interactions needed to inject a new reachability pattern.

The inaccuracy of the measured injection time is very small (65 ms on average) and does not contribute to any detectable effects in our measurements. Details of the inaccuracy and other limitations of our emulated reachability patterns are discussed below.

Inaccuracy Let I_i denote the injection time of the i^{th} injection event. Let δ_s denote the setup latency, which is the time from beginning a setup phase and until all

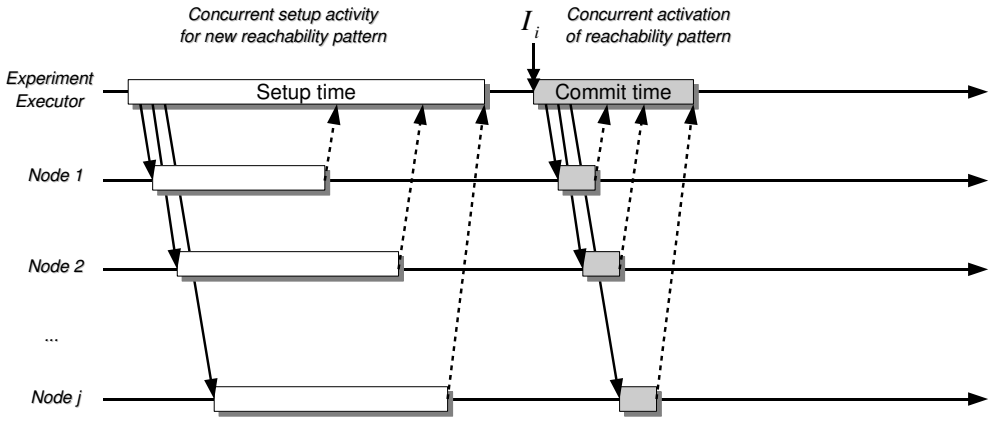


Figure 13.4: Illustration of the setup and commit phases.

nodes have been configured. Let δ_c be the commit latency, which is the time from the injection time I_i until all nodes have activated the new reachability pattern. These latencies limits the accuracy that can be obtained, as illustrated in Figure 13.5. Two consecutive injections are shown, I_1 followed by I_2, which serves to illustrates the smallest possible delay between a pair of injections. That is, $\delta_s + \delta_c$ is the minimum time between two consecutive injections. Furthermore, δ_c limits the accuracy in detection of a newly injected reachability pattern. This is since each node may perceive the new reachability at different times, at most separated by δ_c.

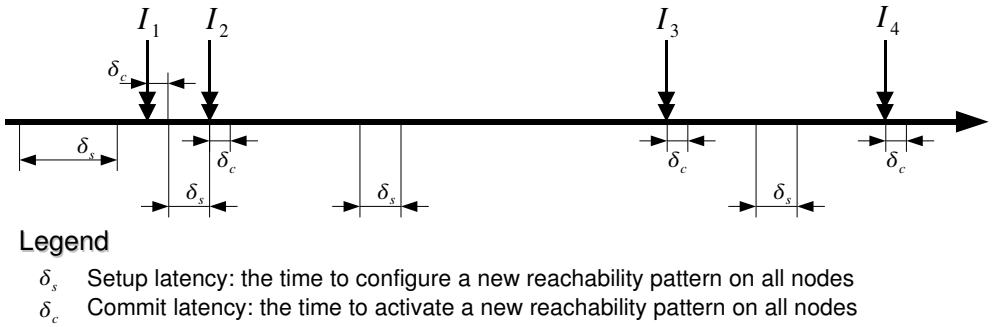


Figure 13.5: Example injection timeline

Table 13.2 provides statistics for these two limiting factors. Note that these statistics do not show the complete picture, since there is an apparent bimodality in the setup

Table 13.2: Statistics for setup and commit latencies for injections (milliseconds).

	Mean	**StdDev**	**Max**	**Min**
δ_s	661.45	217.98	1040	425
δ_c	64.88	27.92	144	35

latency, as illustrated in Figure 13.6. The peak around 900 ms stems from the first setup phase shown before I_1 in Figure 13.5 and is due to connection establishment between the experiment executor and the node-local fault injector modules. The peak around 450 ms is the latency typically seen between injections I_2, I_3 and I_4. Thus, taking also the commit latency (65 ms on average) into account, these observations seem to indicate that a pair of consecutive injections that arrive within an interval shorter than 500-600 ms cannot be reliably tested using our fault injector. However, such close injection events are very rare, and in most cases would not have been detected by the Jgroup failure detector as a network partition in the first place.

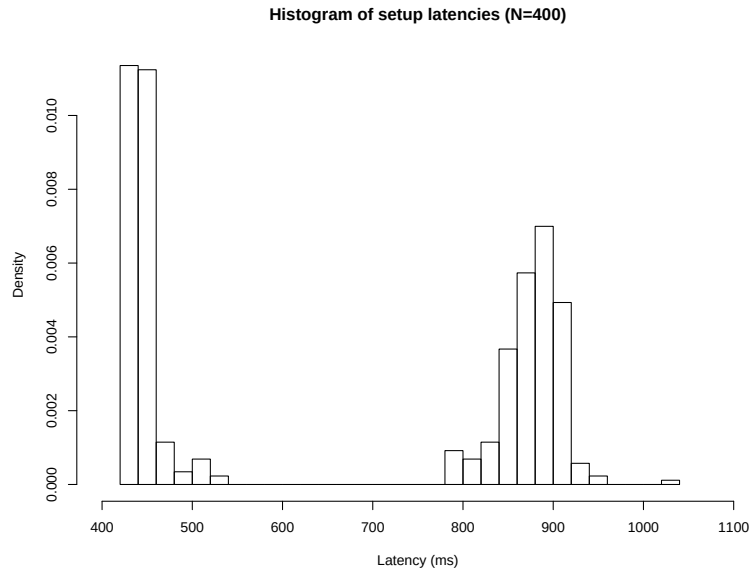


Figure 13.6: Histogram of setup latencies

The density of the commit latency (δ_c) is included in Figure 13.7. The variations in the commit latency are rather small, and are most likely due to correlation between

the commit invocations and the garbage collection mechanism of the various JVMs in the target system. It is the commit latency that limits the accuracy of partition detection. Hence, the results presented in Chapter 16 may have an inaccuracy of approximately 65 ms on average.

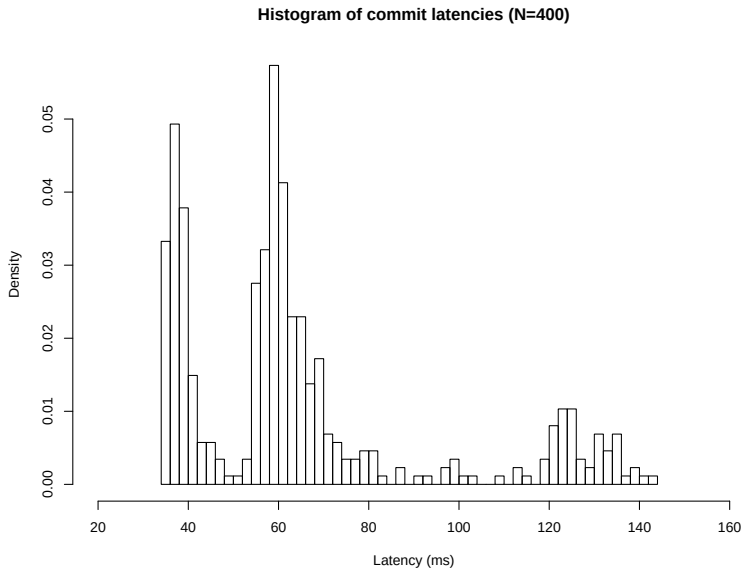


Figure 13.7: Histogram of commit latencies

13.4.3 Improvements to Avoid Code Modification

Modifying the original source code to insert instrumentation logic is a common approach to evaluate systems, and is also used in the framework presented herein. The drawback is that it makes the code harder to understand – it is difficult to determine what is evaluation logic (fault injector) and what is actual system algorithms. Moreover, the evaluation logic must be removed or disabled when a real system is deployed.

A new programming technique called aspect-oriented programming (AOP) [31] has recently become popular. AOP extends object-oriented programming by introducing a new unit of modularity, called an *aspect*. Aspects are special modules that focus on *crosscutting concerns* in a system that are difficult to address in traditional object-oriented languages.

In the future, code instrumentation could instead be handled using AOP techniques. That is, it is possible to define aspects that can ”insert“ logging statements or other interception logic at certain points in the code without having to modify the actual source code of the system. Such aspects are typically used only during testing, and can be easily be removed from a deployed system since they are provided by separate modules. An example could be an aspect to emulate network partition scenarios implemented through packet discarding. Such code is hardly useful in a deployed system, and if implemented as an aspect (a separate module), it is much easier to remove from the system than modifying the original source code.

Note that using AOP techniques for inserting code and intercepting method calls may result in a slightly higher overhead compared to inline code instrumention as currently used.

13.5 Experiment Analysis

Experiment analysis is specific to the kind of study being performed, and is organized in two separate modules:

- Experiment analysis module (EAM)
- Study analysis module (SAM)

With EAM each experiment is processed individually; it can be used to extract measurement information from the log files (event traces) obtained after the completion of the experiment. Event traces from all nodes in the target system are collected and may be used to construct a single global event trace (timeline). This is done for the studies in Chapter 15 and Chapter 16.

The purpose of SAM is to aggregate the results obtained from the individual experiments to compute various statistical properties for the evaluation. Hence, SAM is used after the completion of all the experiments in the study. Typically, EAM is reused for data collection, by extracting relevant measures for which statistics should be computed. For example to extract detection and recovery delays, or to determine the system down time.

In general, two kinds of studies are considered:

- Studies for system validation and error correction.
- Studies for performance and dependability evaluation.

The former aims to test the system functionality and allow the developer to obtain debug logs that can be used for debugging and error correction. In this case, EAM can be used online during a study execution to analyze each experiment individually after their completion. The results of the analysis can be used by the experiment executor to make decisions about the continued execution of the study. This is useful to determine whether a particular experiment should be repeated or if the study should be terminated. An experiment may be repeated by using the same fault injection data as in the original execution of the experiment; recall that fault injections may occur at randomized times. Repeating an experiment in this manner is useful to obtain additional debug logs from similar experiments, to better understand the incorrect system behavior and to be able to debug the problem. Furthermore, after a fix has been applied, the same fault injection scenario can again be repeated to determine if the problem has been solved.

Note that repeating the same fault injection scenario does not guarantee that the same bug is revealed again. However, if a particular bug is revealed in repeated experiments prior to applying a fix, and not after the fix, increased confidence is gained that the bug has in fact been fixed. After fixing the bug, a full randomized fault injection test should be performed again to determine if the fix has introduced new bugs.

Two studies focusing on performance and dependability evaluation are provided in Chapter 15 and Chapter 16.

13.6 Summary

The experiment framework presented in this chapter has proved exceptionally useful in uncovering at least a dozen subtle bugs in the Jgroup/ARM platform, allowing systematic stress and regression testing. Below the impact of instrumentation and injection accuracy is discussed.

Impact of the instrumentation code The experiment framework relies on logging system events to memory during experiment execution. Generally, these events are infrequent and thus will not influence the overall system significantly. Periodically, events are flushed to disk and this may result in minor disturbances if disk access is congested.

Crash failure injections are passive in that they are only activated at the injection time, thus there is no other impact on the system during an experiment.

On the other hand, partition failure injections are implemented by discarding packets according to the configured reachability pattern. With this approach some minor processing at each node is required, even for packets from ”disconnected“ nodes. This processing is done for all packets, independent of the reachability pattern. The processing overhead for each packet is very low. However, given a high system load, this packet processing overhead may have an impact on system performance.

The Loki fault injector used to evaluate correlated network partitions in the Coda filesystem [71] is different from our approach. Instead of inline packet discarding, a firewall mechanism was used to configure blocking on specific ports. This approach is likely to give slightly less overhead as packets are discarded by the operating system. However, the drawback with this approach is that configuring the firewall often requires administrator (root) access.

Unexpected system behaviors due to the instrumentation code have not been observed in our measurements in Part IV.

Injection accuracy The accuracy obtained from partition failure injections is very good. In the worst case a delay of 144 ms (the max commit latency) may separate the activation of a particular reachability pattern at two nodes. Hence, nodes may perceive a different reachability pattern at the same time instance. The impact of such a small delay is insignificant, since the view agreement protocol takes much longer to complete in most cases. In the worst case, it could cause additional runs of the protocol. The fact that different nodes perceive a different reachability pattern at the same time instance may also occur in real disconnection scenarios, e.g. if routing tables have been incorrectly altered. Such errors should be tolerated by the middleware platform.

Chapter 14

Client-side Update Measurements

In this chapter the client-side updating techniques discussed in Chapter 9 are evaluated experimentally. The techniques are aimed at maintaining an approximate client-side membership consistency with the dynamic server-side group membership. Such techniques are especially important when used in conjunction with ARM, since the installation of new replicas to replace failed ones leads to client-side inconsistency. Each technique is evaluated by performing client invocations on a group of two servers, and measuring the latencies observed by clients in the various phases of the techniques.

The main objectives of the experiments are:

1. To determine the performance impact on the anycast load balancing, and
2. to investigate the client update time, T_{cu}, of the techniques.

In addition, we are also interested in the client-side failover latency, T_f, and the total update delay, T_u. The latter includes the server-side recovery time, T_r.

Section 14.1 explains the experiment setup for evaluating the client update techniques. Section 14.2 presents and discusses the results of the experiments. Section 14.3 concludes the chapter.

14.1 Experiment Setup

Three experiments were conducted, using a cluster of eleven P4 2.4 GHz Linux machines interconnected over a 100 Mbit/s LAN. In each experiment, a two-way replicated echo server were initialized using the ARM framework, allowing recovery from server failures. The replication policy of the echo server is configured to use a 3 second safety margin before activating recovery (see Section 11.5), and its redundancy level is $R_{\text{init}} := 2$ initially, and $R_{\text{min}} := 2$ is the minimal redundancy level to be maintained by ARM.

Figure 14.1 gives an overview of the experiment configuration, consisting of some 56 clients (distributed over seven physical machines), each of which perform a continuous stream of *anycast* method invocations. The method being invoked takes an array of 1000 bytes as argument and returns the same array. Each of the two servers handle their fair share of the client requests through the anycast load balancing mechanism. After reaching a steady state performance level, one of the servers is crashed manually. The remaining server then notifies ARM of the failure, eventually causing ARM to install a replacement server on the spare node.

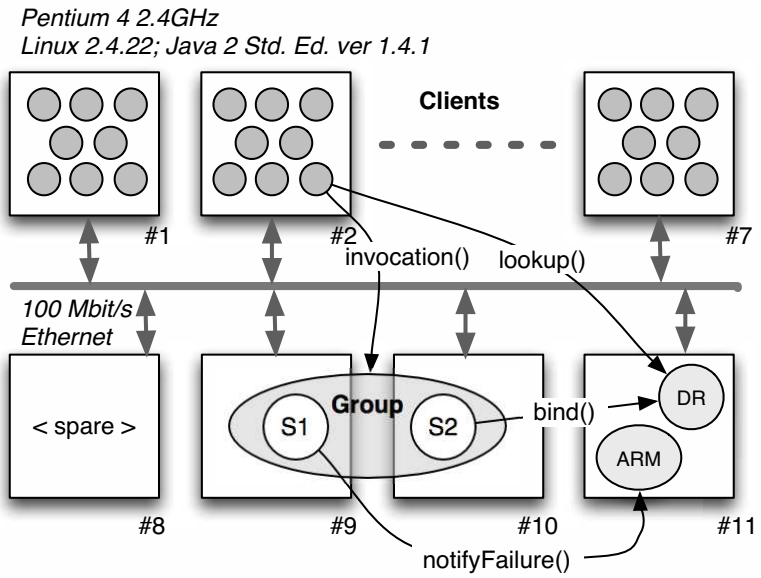


Figure 14.1: Experiment setup.

14.2 Client-side Update Measurements

In each experiment the round trip invocation-response latency is measured at the clients. The time given on the abscissa in the plots corresponds to the receive time of an invocation and hence, a long delay in service provision, e.g. due to a server crash, during an invocation will appear as a blank period on the abscissa. The plots shows one dot per observation of the invocation-response latency along with the mean value for all observations as a function of the time on the abscissa. The vertical lines in the plots are used to indicate the occurrence time of relevant events.

14.2.1 No Update

The no update approach is included for reference, and serves to demonstrate the benefit of using the other techniques. Figure 14.2 shows the invocation-response latency before and after the server crash without using a client-side update mechanism. That is, the client-side proxy do not attempt to update its references to include the recovered server.

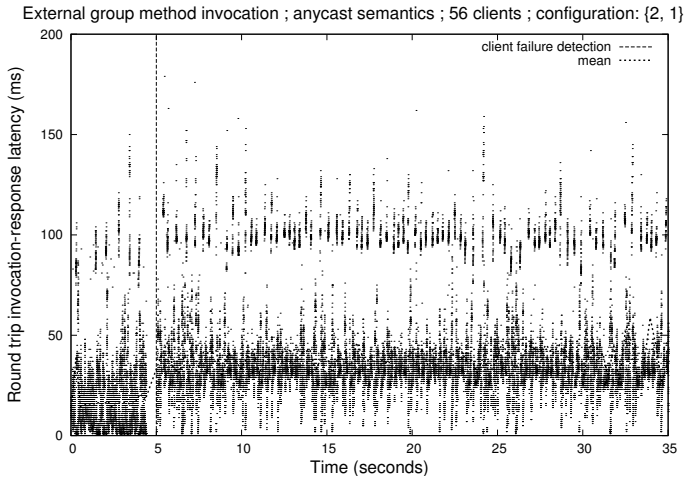


Figure 14.2: No update.

Initially, the client-side proxy holds a reference to both server endpoints, and will load-balance client invocations on both servers. However, after the crash, only a single server is know to the client-side proxy and hence it is unable to exploit the

recovered server. For this reason, the response time for invocations increases from about 19 ms to about 38 ms since the load doubles on the remaining server (known to the client). The blank period immediately before the 5 second mark due to the server crash and is discussed in the next section.

The no update approach is also demonstrated in Section 9.2, where the client invocation performance degrades without updating the client-side, and eventually server failures are exposed to the client application.

The observations with higher invocation latencies (approximately 100 ms) are observed at a regular rate of about 0.35 seconds. These invocations take longer to complete due to correlation with server-side garbage collection (GC) performed regularly by the Java virtual machine. In addition to these large variations, there are also other stochastic components that are likely due to request accumulation at the selected server, and client-side GC and context-switching, among other things. These variations are observed in all the following experiments as well.

14.2.2 Client-side View Refresh

Figure 14.3 illustrates the results obtained using the client-side view refresh technique.

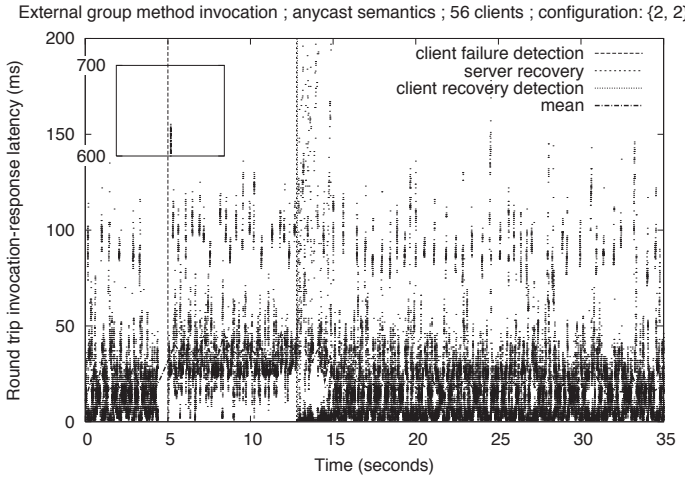


Figure 14.3: Client-side view refresh.

After the server crash (detected by the client at the 5 second mark), the proxy removes its reference to the failed server and continues to use only the remaining server known

to the proxy. The failover latency (T_f) can be seen as the blank period immediately before to the client failure detection point. The average $T_f = 631.8$ ms. Delayed invocations due to failover are shown as an inset, as these delays fall too far outside the plot range, and represents only a minor portion of all invocations.

As the figure show, the actual server recovery instance and the point at which all the clients have become updated is almost overlapping (difference of $T_{cu} = 52$ ms). The total client update delay is $T_u = 7.82$ seconds, where server recovery time contributes $T_r = 7.77$ seconds. The increase in invocation latency immediately after recovery, stems from connection establishment. For readability, not all data related to post recovery is shown in the figure; there are also some invocation latencies in the range $[200, 400]$.

The system have a steady state performance between failure and recovery in Figure 14.3 that is identical to the case with no update. However, once the client detects the replacement server, the steady state performance returns to the default level. However, notice that there is a period of approximately 2.1 seconds before all clients have established a connection with the replacement server.

14.2.3 Periodic Refresh

Figure 14.4 shows the results of the periodic refresh technique.

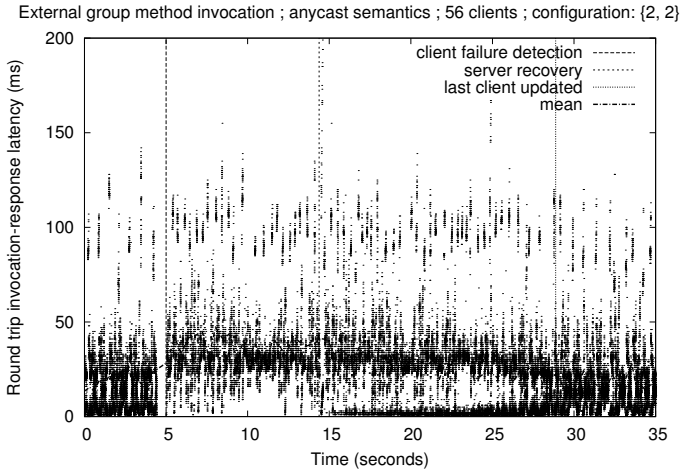


Figure 14.4: Periodic refresh.

In this experiment, we used a refresh rate of 15 seconds. As in an operative system, the clients start independently and asynchronously. Hence refreshes will occur uniformly distributed over the refresh interval and the clients gradually establish connections to the replacement server. This leads to an average client update time as high as $T_{cu} = 8.16$ seconds. The total update delay is $T_u = 28.88$ seconds.

14.3 Summary of Findings

In Chapter 9 and in this chapter, a potential performance bottleneck and other limitations in the client-side proxy and the dependable registry are identified. These limitations may lead to significantly larger invocation latencies for clients, and may render server failures visible to the client application, an undesirable property in middleware frameworks. Several techniques have been proposed for maintaining consistency between the group membership and its representations within the dependable registry and residing at clients. A performance study has been carried out to reveal the impact of inconsistent client-side proxies and to compare our proposed techniques. The client-side view refresh technique has been shown to be the most effective approach.

Chapter 15

Measurement based Crash Failure Dependability Evaluation

Group communication and fault treatment systems for development of dependable distributed applications have received considerable attention in recent years [87, 41, 97, 3, 104, 102]. Assessing and evaluating the dependability characteristics of such systems, however, have not received an equal amount of attention.

This chapter presents an extensive evaluation of crash failure and recovery behavior of the Jgroup/ARM middleware framework [87, 14, 80]. The evaluation approach is based on *stratified sampling* combined with *fault injections* for estimating the dependability attributes of a service deployed using the Jgroup/ARM middleware framework. The experimental evaluation is performed focusing on a service provided by a triplicated server, and indicative predictions of various dependability attributes of the service are obtained. The evaluation shows that a very high availability and MTBF may be achieved for services based on Jgroup/ARM. The main principles of the dependability evaluation technique presented herein is due to B. E. Helvik, and the results presented in this chapter were published in [57].

This chapter is structured as follows: Section 15.1 introduces the evaluation technique and relates it to previous works. Section 15.2 describes the target system for our measurements, while Section 15.3 presents the measurement setup and strategy, together with the associated estimators for dependability attributes. Experimental results are provided in Section 15.4, and finally Section 15.5 gives concluding remarks.

15.1 Introduction

In the evaluation, dependability attributes are predicted through a *stratified sampling* [72] approach. A series of experiments are performed; in each experiment, one or more faults are injected according to an accelerated homogeneous Poisson process. The approach defines strata in terms of the number of near-coincident failure events that occur in a fault injection experiment. By near-coincident is meant failures occurring before the previous is completely handled. Hence, *a posteriori stratification* is performed where experiments are allocated to strata after they have been carried out. This as opposed to the more common prior stratification where strata are defined before the experiment [34]. Three strata are considered, i.e. single failures, and double and triple near-coincident failures. The nodes of the system under study is assumed to follow the crash failure semantics. For the duration of an experiment, the events of interest are monitored, and post-experiment analysis is performed to construct a single global timeline of fault injections and other relevant events. The timeline is used to compute trajectories on a predefined state machine.

Depending on the number of injected faults, each experiment is classified into one of the strata, and various statistics for the experiments are obtained. These statistical measures are then used as input to estimators of dependability attributes, including *unavailability*, *system failure intensity* and *down times*. The approach may also be used to find periods with reduced performance due to fault handling. An additional benefit of this thorough evaluation is that the fault handling capability of Jgroup/ARM has been tested extensively, enabling the discovery of rarely occurring implementation faults of both the distributed service under study and the Jgroup/ARM framework itself.

Fault injection is a valuable and widely used means for the assessment of fault tolerant systems, see for instance [4, 5, 52]. Previously, stratified sampling has been used in combination with fault injection experiments to estimate fault tolerance coverage, as presented in [34]. Furthermore, for testing specific parts of a system, fault injection triggers has been used on a subset of the global state space [28]. These approaches are very useful in testing and evaluating specific aspects of a system. However, our objective is to perform an overall evaluation of the system and its ability to handle node[1] failures and hence, random injections of crash failures in a operational system and post stratification is applied.

[1] In [57] we used the term *processor*, while in this chapter the term *node* is used to be consistent throughout the dissertation.

Delta-4 [102] provide fault treatment mechanisms similar to those of ARM. Fault injections were also used in Delta-4 [12], focusing on removal of design/implementation faults in fault tolerance mechanisms. However, we are not aware of reports on the evaluation of the fault treatment mechanisms in Delta-4, comparable to those presented herein. The fault injection scheme used in this work, combined with post-experiment analysis also facilitate detection of implementation faults, and in addition allows for systematic regression testing.

The AQuA [104, 103] framework is based on CORBA and also support fault treatment. Unlike Jgroup/ARM, it does not deal with partition failures and relies on the closed group model [64] which limits its scalability with respect to supporting a large number of groups. The evaluation of AQuA presented in [103] only provide the various delays involved in the recovery time. In this paper, focus is on estimating dependability attributes of services deployed through ARM.

15.2 Target System

Figure 15.1 shows the target system for our measurements. It consists of a cluster with a total of $n = 8$ identical nodes, initially hosting a single server replica as shown. In the experiments, ARM uses the distribution policy described in Section 10.1.1. It will avoid co-locating two replicas of the same type, and at the same time it will try to keep the replica count per node to a minimum. Different services may share the same node.

The ARM infrastructure (i.e. the RM group) is located on nodes 1-3. Nodes 5-7 host the *monitored service* (MS), while nodes 4 and 8 host the *additional service* (AS). The latter was added to assess ARM's ability to handle multiple concurrent failure recoveries at different groups, and to provide a more realistic scenario. Finally, an external machine hosts the *experiment executor* that is used to run the experiments as discussed in Chapter 13. The replication policy (see Section 10.1.2) for all the deployed services requires that ARM tries to maintain a fixed minimal redundancy level for each of the three services as follows: R_{min} (RM) := 3, R_{min} (MS) := 3 and R_{min} (AS) := 2. Hence, the RM group is at least as fault-tolerant as the remaining components of the system.

In the following, we will focus our attention on the MS service, that constitutes our *subsystem of interest*. This subsystem will be the subject of our observations and measurements, with the aim of predicting its dependability attributes. Note that focusing

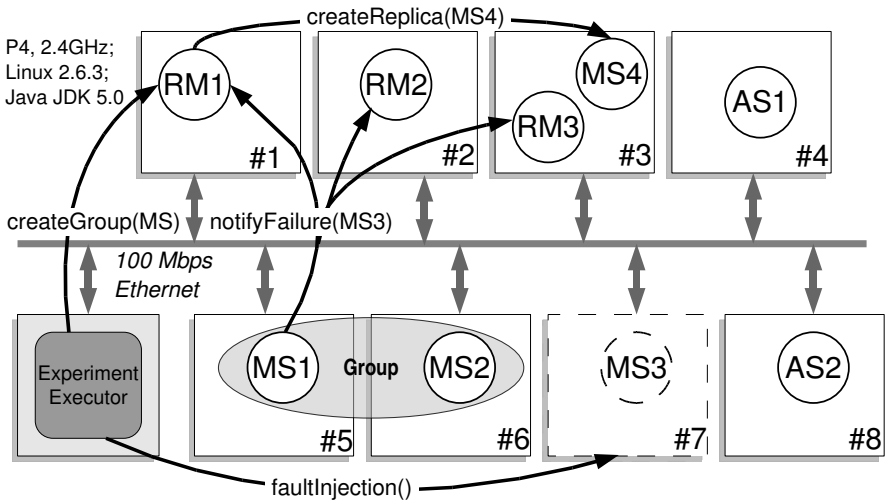


Figure 15.1: Target system illustrated.

on a particular subsystem of interest is for simplifying the presentation. Observations of several subsystems could be done simultaneously and estimates/predictions of all services and the ARM infrastructure may be obtained during the same experiment.

15.2.1 The State Machine

There is a set of states which we can observe and which are sufficient to determine the dependability characteristics of the service(s) regarded. Note that these are not all the operational states of the complete system, but the set of states associated with the MS service. Thus, the failure-recovery behavior of the MS service can be modeled according to the *state machine* in Figure 15.2, irrespective of the states of the ARM and AS subsystems.

The state machine is not used to control fault injections based on triggers on a subset of the global state space as in [28]; instead it is only used offline during a posteriori analysis of fault injection experiments based on random sampling. In the analysis the independent event traces collected from the target system nodes are merged into a single global timeline of events, which corresponds to an approximation of the actual state transitions of the whole system. The events in the global trace corresponds to the events of the state machine in Figure 15.2. Given this global event trace, we can compute the trajectory of visited states and the time spent in each of the states.

These trajectories allow us to classify the experiments and to estimate a number of dependability attributes for the monitored service.

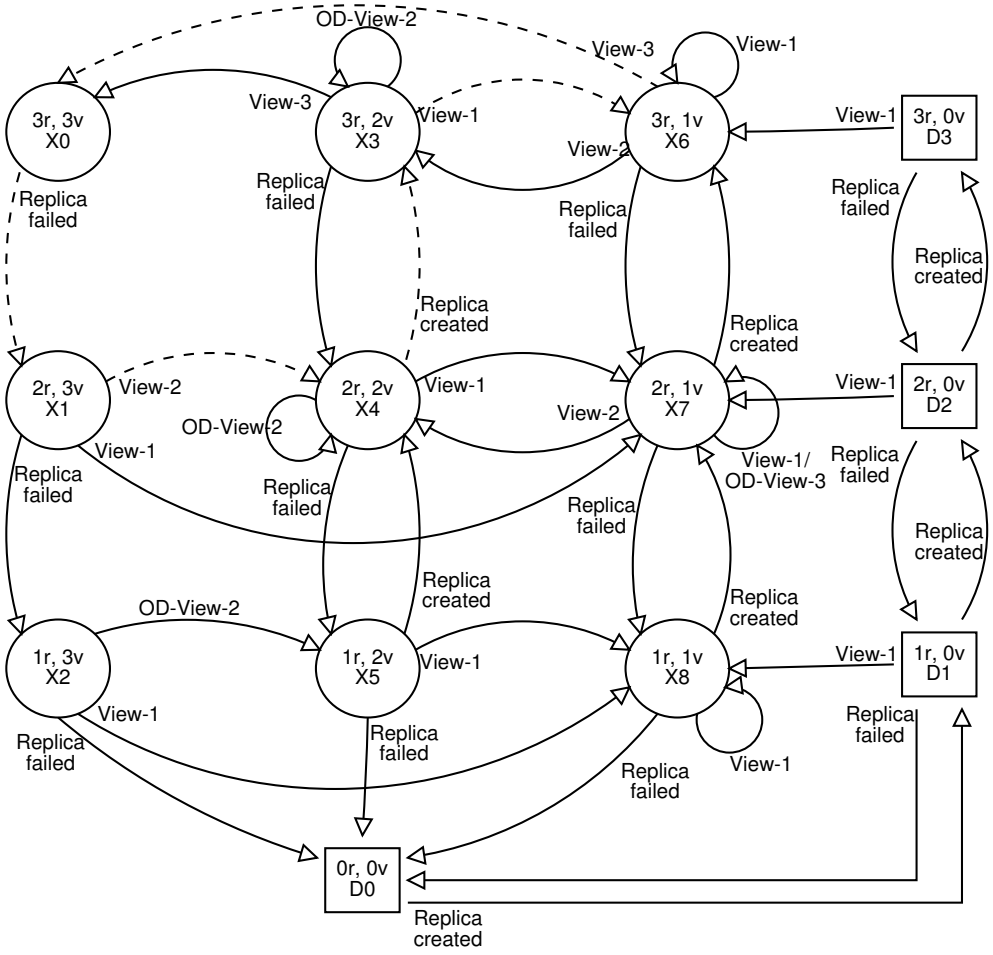


Figure 15.2: State machine illustrating a sample of the possible state changes of the MS service being measured.

Server-side availability is define in terms of view installations; this is since a view event confirms that the servers contained in the view are all ready to provide service to clients. A service is defined to be *unavailable* (squared states) if none of the group members have installed a view, and *available* (circular states) if at least one member has installed a view. Each *up state* is identified by $X\sharp$ and a tuple $(x\mathbf{r}, y\mathbf{v})$, where x is the number of installed replicas ($\mathbf{r}$), and y is the number of members in the current

view ($\mathbf{v}$) of the server group. *Down states* are identified by $D\sharp$.

In the state machine, only events that may affect the availability of the service are considered, such as *view changes*, *replica creations* as seen from the perspective of ARM, and *replica failures* as perceived by the corresponding MS nodes that fails. View changes, in particular, are denoted by View-c, where c is the cardinality of the view. In addition, *fault injection* events may occur in any of the states, however for readability they are not included in the figure.

As a sample failure-recovery behavior, consider the trajectory composed of the state transitions with dashed lines, starting and ending in the $X0$ state. This is the most common trajectory. For simplicity the View-c events in the state machine reflect the series of views as seen by ARM, and do not consider the existence of concurrent views. So, after recovering from a failure (e.g. moving from state $X4$ to state $X3$), the newly created member will install a singleton view and thus be the leader of that view, sending a View-1 event to ARM (from state $X3$ to state $X6$). Only after this installation (required by the view synchrony property) a View-3 event will be delivered to ARM, causing a transition from state $X6$ to $X0$. The above simplification does not affect the availability of the service. It is assumed that client requests are only delayed during failure-recovery cycles as long as the service is in an operational state [78]. Such delays are not considered part of the availability measure as opposed to [63]. Further analysis of these client perceived delays is deferred for future work.

Note that some of the states have self-referring transitions on view change events. These are needed for several reasons, one being that the ARM framework may see view change notifications from several replicas, before they have formed a common view. In addition, ARM will on rare occasions receive what we call *outdated views* (OD), that are due to minor inaccuracies in our measurements. For instance, a View-3 event may occur while in state $X7$. This can occur if at some point we are in the $X6$ state, when a group member sends out a notification of a View-3 event, and shortly after another member of that group fails and logs a *Replica failed* event. However, given that the View-3 event is still in the "air", and has not yet been logged, the *Replica failed* event will appear to have occurred before the View-3 event in the global trace. To compensate for this behavior, we have inserted additional View-c transitions, prefixed by OD, in some of the states.

Also note the View-2 transition from $X2$ to $X5$. This is also due to an outdated view and can occur if ARM triggers recovery on the View-2 event before receiving a View-1 event. Note that the state transitions in Figure 15.2 may not be complete as presented, however, no other transitions have been observed during our experiments.

In the following, we will assume that the service has been initialized correctly into state $X0$, and thus we do not consider the initial transitions leading to this state.

15.3 Measurements

This section gives motivation for our measurement approach. Furthermore, we discuss in detail the sampling scheme used to assess the fault handling capability of the Jgroup/ARM framework and to provide input for the estimators of dependability attributes.

15.3.1 Experiment Outline

In each experiment, one or more faults are injected. The failure injection pattern is as if it emerged from a Poisson process. There may be multiple near-coincident crash failures before the system stabilizes, i.e. a new failure may be injected before the previous has been completely handled. This will "simulate" the rare occurrence of nearly coincident failures which may bring the service down. The Poissonian character of the injected failures is achieved through generation of fault injection times and the selection of the set of nodes in which to inject faults, according to a uniform distribution. See the Sampling Scheme in Section 15.3.2 on how this yields a Poisson fault process. Nodes to crash are drawn from the entire target system. Hence, the injected faults may affect the ARM infrastructure itself, the monitored subsystem (MS) or the additional service (AS), all of which are being managed by the ARM framework. However, only state trajectories for the monitored subsystem are computed, and these are used for predicting various dependability attributes of MS. A beneficial "side-affect" of this sampling scheme is that it has shown to be very useful with respect to performing extensive testing of the fault handling capabilities of the Jgroup/ARM. During previous experiments several design and implementation faults have been revealed. In each experiment, at most $k = 3$ fault injections are performed. Since all nodes in the target system have allocated replicas initially, failures will cause ARM to reuse nodes as shown in Figure 15.1, where the replica of node 7 is recreated at node 3.

15.3.1.1 Time Constants Considered

Assuming services are deployed using the ARM framework, the crashed nodes will have a *node recovery time* (T_{NR}) which is much longer than the *service recovery*

time (T_{SR}). Further, we assume that the nodes will stay crashed for the remaining part of the experiment. In other words, a service replica will typically be restarted on a different node as soon as ARM concludes that a node crash has occurred. However, the time until the nodes are recovered, is assumed to be negligible compared to the *time between failures* (T_{BF}) in a real system. Thus in the predictions it is assumed that the occurrence intensity of new trajectories (i.e. first failure in a fully recovered system) is $n\lambda$, neglecting the short interval with a reduced number of nodes between T_{SR} and T_{NR}. Figure 15.3 shows these relations, starting with the first failure event t_{i_1}. Furthermore, there will be no resource exhaustion, i.e. there are sufficient nodes to execute all deployed services, including the ARM infrastructure.

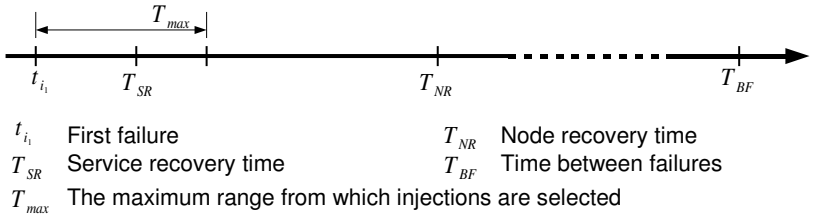


Figure 15.3: The relation between the service and node recovery periods and the time between failures.

15.3.1.2 The Failure Trajectory

A failure trajectory is the series of events and states of the monitored subsystem following the first node failure and until all the concurrent failure activities have concluded and all subsystems are recovered and fully replicated. The trajectory will always start and end in state $X0$ (see Figure 15.2). *If* the first node failure affects the monitored service, it causes it to leave its steady operational state $X0$ and *if* it is the last service to recover, we will see a return to the same state like in Figure 15.4.

We denote the j^{th} event in the i^{th} trajectory by i_j, the time it takes place by t_{i_j} and the state after the event by X_{i_j} (corresponds to the states in Figure 15.2). Note that all relevant events in the system are included, and a failure or another event does not necessarily cause a change of state in the monitored subsystem. For instance, the failure of a node which supports only the ARM or AS subsystems, will not necessarily result in a change of state in the MS service, but it is likely that it will influence the

handling of immediately preceding or succeeding failures affecting the service. Let $X_i(t)$ denote the state of the MS service at time t in the i^{th} failure trajectory,

$$X_i(t) = \begin{cases} X_{i_j} & t_{i_j} < t \leq t_{i_{j+1}},\ j = 1, \ldots, m_i \\ X0 & \text{Otherwise} \end{cases}$$

where m_i is the last event of the i^{th} trajectory before all concurrent failure activities have concluded, and all subsystems are fully replicated. During the measurements, a trajectory sample is recorded as the list

$$\underline{X}_i = \left\{X0, t_{i_1}, X_{i_1}, t_{i_2}, X_{i_2}, t_{i_3}, \ldots, t_{i_{m_i}}, X0\right\}.$$

Trajectories for which the MS service does not leave the $X0$ state are also recorded.

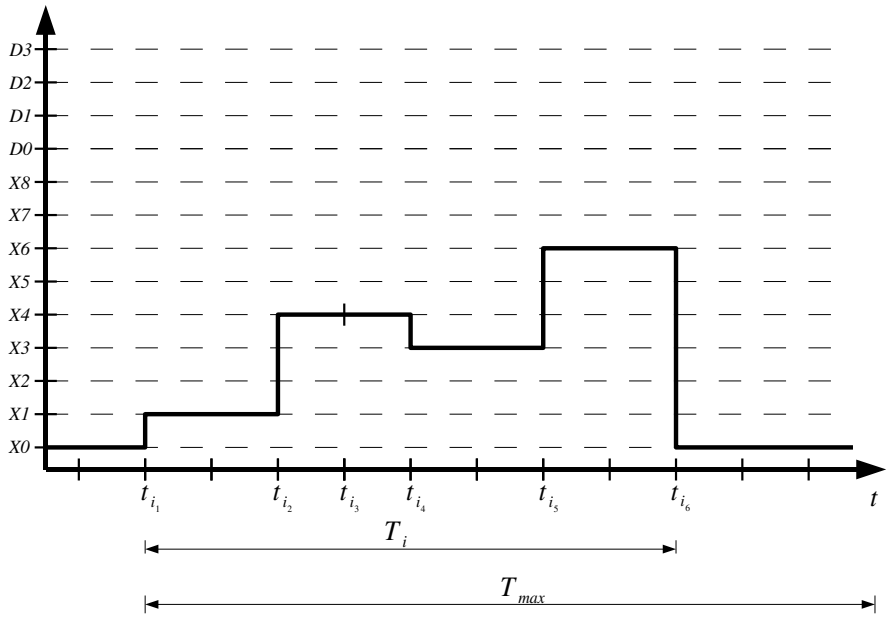


Figure 15.4: Sample failure trajectory, where all but failure event t_{i3} affects the subsystem of interest. This is the most common failure trajectory, cf. the dashed line in Figure 15.2.

15.3.1.3 Characteristics Obtained from a Failure Trajectory

The unknown probability of failure trajectory i is p_i. For brevity we denote the duration of trajectory i by $T_i = t_{i_{m_i}} - t_{i_1}$, and its expectation $\Theta = E(T) = \sum_{\forall i} p_i T_i$. In the following, let Y_i denote a sample from the experiment. The sample may be obtained from the trajectory by some function g, i.e. $Y_i = g(\underline{X}_i)$. The duration of a trajectory presented above may serve as an example. Determining the dependability attributes of the system are other possible samples that can be extracted from the experiment data. To determine these, it is assumed that the failure rate in the $X0$ state is $n\lambda$, that the expected sojourn time in this state is much longer than the expected trajectory duration, and that a particular trajectory is independent of previous trajectories.

Unavailability The time spent in a down state during a trajectory i is given by

$$Y_i^d = g(\underline{X}_i) = \sum_{j=1}^{m_i} I(X_{i_j} \in \mathfrak{F})(t_{i_{j+1}} - t_{i_j}), \tag{15.1}$$

where $I(\cdots)$ is the indicator function and $\mathfrak{F} = \{D0, D1, D2, D3\}$ is the set of down states (the squared states in Figure 15.2). Given that the periods in state $X0$ (the OK-periods) alternate with the failure trajectories, and are independent and much longer than the failure trajectory periods, we can obtain a measure for the service unavailability

$$\hat{U} = \frac{E(Y^d)}{E(Y^d) + (n\lambda)^{-1}} \approx E(Y^d)n\lambda. \tag{15.2}$$

Note that the collective failure intensity of all nodes when there are no faults in the system, is only marginally different from the intensity of trajectories. The difference is due to the restoration of failed nodes during a trajectory, and is negligible.

Probability of failure, reliability In this case, let $Y_i^f = 1$ if trajectory i visits one or more down states, otherwise let $Y_i^f = 0$.

$$Y_i^f = g(\underline{X}_i) = I(\exists X_{i_j} \in \mathfrak{F})_{j=1,\ldots,m_i}. \tag{15.3}$$

Disregarding multiple down periods in the same trajectory and assuming that system failures are rare, it is found that the system failure intensity is approximately

$$\hat{\Lambda} = \frac{1}{\text{MTBF}} \approx \frac{E(Y^f)}{E(Y^d) + (n\lambda)^{-1}} \approx E(Y^f)n\lambda. \tag{15.4}$$

The predicted reliability function $\hat{R}(t) = \exp(-\hat{\Lambda}t)$ may be obtained. The system failure process will be close to a Poisson process since the trajectories starts according to a Poisson process and each trajectory will with an (unknown) constant probability result in a system failure, i.e. we have a splitting Poisson process [40, Ch. 5.3.2]. In addition, the mean down time MDT $= \hat{U}/\hat{\Lambda}$ may be obtained. MDT and the down time distribution may of course also be measured directly from the trajectories visiting the set of down states.

The above examples are chosen for illustration and the assumptions made for simplicity. By introducing rewards [106] associated with the states and transitions, we may obtain predictions of far more comprehensive performability measures of the system.

15.3.2 Experimental Strategy

The experimental strategy is based on a *post stratified random sampling* approach. For an introduction to stratified sampling see for instance [72]. This section elaborates on how the experiments are classified in different strata, and how the sampling is performed.

15.3.2.1 Stratification

Only some of the events along a failure trajectory will actually be failure events. The first event of each trajectory will always be a failure, and in a typical operational environment usually the only one. However, in the experiments we consider also multiple near-coincident failures which may require concurrent failure handling. In considering such failure scenarios, our experimental strategy is based upon subdividing the trajectories into strata $\mathcal{S}_k$ based on the number of failure events k in each of the trajectories. Each of the strata are sampled separately, and the number of samples in each stratum are random variables determined a posteriori. This is different from previous work [34] in which the number of samples in each stratum is fixed in advance.

An example failure trajectory reaching stratum $\mathcal{S}_3$ drawn from the experiment data is shown in Figure 15.5. Three near-coincident faults were injected in this particular experiment. The first and last failure affect the MS service, while the second affect the RM service. The RM failure and its related events, as indicated on the curve, do not cause state transitions in the state machine (see Figure 15.2) of the MS service.

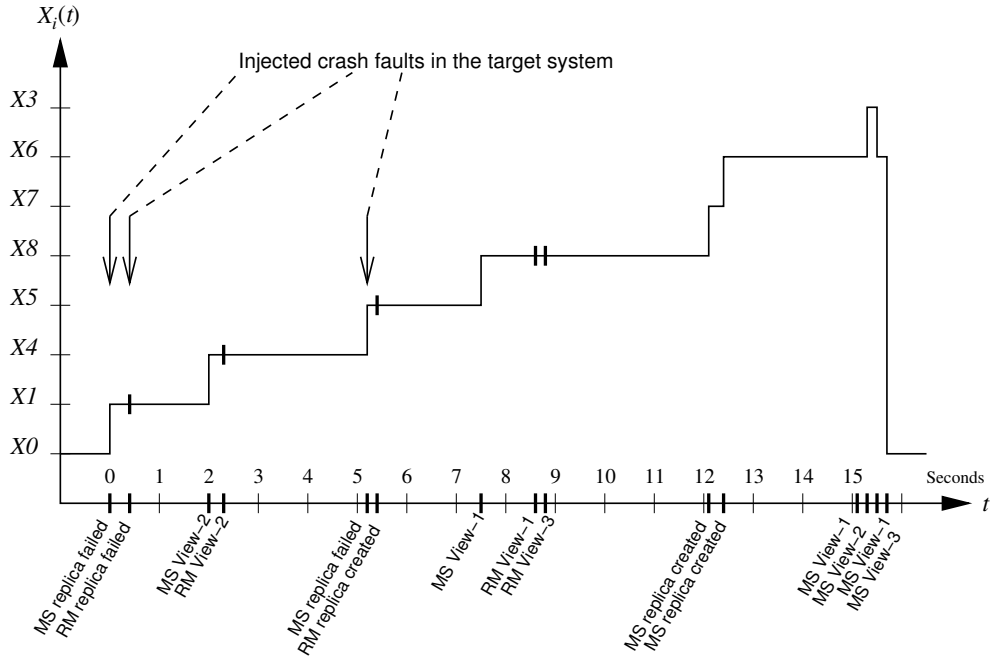


Figure 15.5: Sample failure trajectory reaching stratum $\mathcal{S}_3$ drawn on an approximate time scale. Only two of three injected faults affect the MS service. The second fault injection affect the RM service.

The collected samples for each stratum are used to obtain statistics for the system in that stratum, e.g. the expectation $E(Y|\mathcal{S}_k)$. The expectation and the variance of the length of the trajectory within a stratum $\mathcal{S}_k$ are denoted $\Theta_k = E(T|\mathcal{S}_k)$ and $\sigma_k = Var(T|\mathcal{S}_k)$, respectively. Estimates may then be obtained by

$$E(Y) = \sum_{k=1}^{\infty} E(Y|\mathcal{S}_k)\pi_k \approx \sum_{k=1}^{3} E(Y|\mathcal{S}_k)\pi_k, \tag{15.5}$$

where $\pi_k = \sum_{\forall i \in \mathcal{S}_k} p_i$ is the probability of a trajectory in stratum $\mathcal{S}_k$. Recall that k represents the number of possible concurrent failure events, and in (15.5), we replace ∞ in the summation with 3, since we only consider up to 3 concurrent failure events. Expressions for π_k are derived in Section 15.3.3.1.

If upper and lower bounds for Y exist, and we are able to determine $\pi_k, k > 3$, we

may also determine bounds for $E(Y)$ without sampling the higher-order strata, i.e.,

$$\sum_{k=1}^{3} E(Y|\mathcal{S}_k)\pi_k + \inf(Y)\sum_{k>3}\pi_k \leq E(Y) \leq \sum_{k=1}^{3} E(Y|\mathcal{S}_k)\pi_k + \sup(Y)\sum_{k>3}\pi_k.$$

Since the probability of k concurrent failures is much greater than $k+1$ failures, $\pi_k \gg \pi_{k+1}$, the bounds will be tight, and for the estimated quantities the effect of estimation errors are expected to be far larger than these bounds. The effect of estimation errors is discussed in Section 15.3.3.2.

15.3.2.2 Sampling Scheme

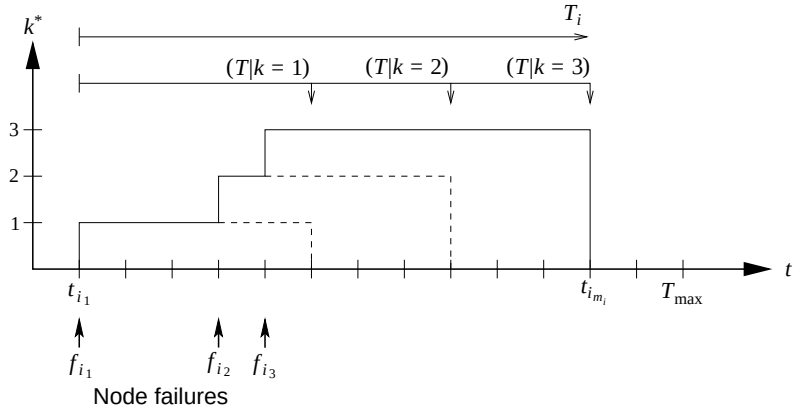
Under the assumption of a homogeneous Poisson fault process with intensity λ per node, it is known that if we have $k-1$ faults (after the first failure starting a trajectory) of n nodes during a fixed interval $[0, T_{\text{max}}\rangle$, these will occur

- uniformly distributed over the set of nodes, and
- each of the faults will occur uniformly over the interval $[0, T_{\text{max}}\rangle$.

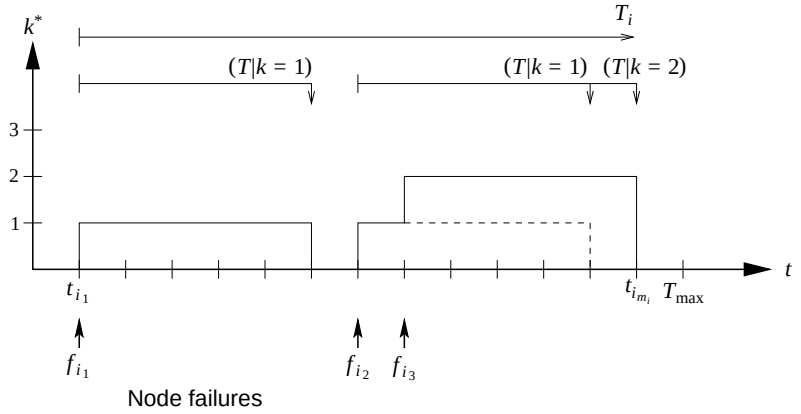
Note that all injected faults *will* manifest themselves as failures, and thus the two terms are used interchangeably. In performing experiments, the value T_{max} is chosen to be longer than any foreseen trajectory of stratum $\mathcal{S}_k$. However, it should not be chosen excessively long, since this may result in too rare observations of higher-order strata.

In the following, let $(T|k=l)$ denote the duration of a trajectory if it completes in stratum $\mathcal{S}_l$, as illustrated in Figure 15.6(a), and let f_{i_l} denote time of the l^{th} failure, relative to the first failure event in the i^{th} failure trajectory. That is, we assume the first failure occur at $f_{i_1} = 0$ and that $f_{i_l} > f_{i_1}, l > 1$. To obtain dependability characteristics for the system, we inject k failures over the interval $[0, T_{\text{max}}\rangle$. This leads to the following failure injection scheme for trajectory i, which *may* reach stratum $\mathcal{S}_k$. However, not all trajectories obtained for experiments with $k > 1$ failure injections will reach stratum $\mathcal{S}_k$, since a trajectory may reach $(T|k=1)$ before the second failure (f_{i_2}) is injected. That is, recovery from the first failure may complete before the second failure injection, as illustrated in Figure 15.6(b). Such experiments contain multiple trajectories, however, only the first trajectory is considered in the analysis to avoid introducing a bias in the results.

The sampling scheme includes the following steps:



(a) Fault injections causing the failure trajectory into higher-order strata.



(b) Failure trajectory that completes before reaching higher-order strata.

Figure 15.6: Sample failure trajectories with different fault injection times.

1. The first failure, starting a failure trajectory i, is at $f_{i_1} = 0$. The following $k-1$ failure instants are drawn uniformly distributed over the interval $[0, T_{\max}\rangle$ and sorted such that $f_{i_q} \leq f_{i_{q+1}}$ yielding the set $\{f_{i_1}, f_{i_2}, \ldots, f_{i_k}\}$. Let k^* denote the reached stratum, and l is the index denoting the number of failures injected so far. Initially, set $k^* := 0$ and $l := 1$.

2. Fault $l \leq k$ is (tentatively) injected at f_{i_l} in node $z_l \in [1, n]$ with probability $1/n$.

   (a) If trajectory i has not yet completed, i.e. $f_{i_l} < T_i$, then set $l := l + 1$ and

      i. If the selected node has not already failed $z_l \notin \{z_w | w < l\}$: *Inject*

fault at f_{i_l} and set $k^* := k^* + 1$

ii. Prepare for next fault injection, i.e. goto 2.

(b) Otherwise the experiment ended "prematurely".

3. Conclude and classify as a stratum $\mathcal{S}_{k^*}$ measurement.

The already failed nodes are kept in the set to maintain the time and space uniformity corresponding to the constant rate Poisson process. Although k failures are not injected in a trajectory, the pattern of injected failures will be as if they came from a Poisson process with a specific number (k^*) of failures during $T_{\max}$. Hence, the failure injections will be representative for a trajectory lasting only a fraction of this time.

15.3.3 Estimators

15.3.3.1 Strata Probabilities

In a real system, the failure intensity λ will be very low, i.e. $\lambda^{-1} \gg T_{\max}$. Hence, we may assume the probability of a failure occurring while the system is on trajectory $i \in \mathcal{S}_1$ is $T_i(n-1)\lambda$. Hence, the probability that a trajectory (sample) belonging to a stratum $\mathcal{S}_k, k > 1$ occurs, given that a stratum $\mathcal{S}_1$ cycle has started is

$$\frac{\sum_{\forall i \in \mathcal{S}_1} p_i T_i (n-1)\lambda}{\sum_{\forall i \in \mathcal{S}_1} p_i} = \frac{\sum_{k>1} \pi_k}{\pi_1}.$$

Due to the small failure intensity, we have that $\sum_{k>1} \pi_k \approx \pi_2$ and the unconditional probability of a sample in stratum $\mathcal{S}_2$ is approximately

$$\pi_2 = (n-1)\lambda \Theta_1 \pi_1. \tag{15.6}$$

This line of argument also applies for the probability of trajectories in stratum $\mathcal{S}_3$. However, in this case we must take into account the first failure occurrence. Let $i \in \mathcal{S}_k \wedge X_i(t_x) \bowtie \mathfrak{f}$ denote a trajectory of stratum $\mathcal{S}_k$, where a failure occurs at t_x. The probability that a trajectory belonging to stratum $\mathcal{S}_k, k > 2$ occurs, given that a stratum $\mathcal{S}_2$ cycle has started is, cf. Figure 15.6(a):

$$\frac{\int \sum_{\forall i \in \mathcal{S}_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i (T_i - t_x)(n-2)\lambda dt_x}{\sum_{\forall i \in \mathcal{S}_2} p_i} = \frac{\sum_{k>2} \pi_k}{\pi_2}. \tag{15.7}$$

Ignoring the constant part of (15.7) for now; the first term on the left-hand side of (15.7) do not depend on t_x and may be reduced as follows:

$$\frac{\int \sum_{\forall i \in \mathcal{S}_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i T_i dt_x}{\sum_{\forall i \in \mathcal{S}_2} p_i} = \frac{\sum_{\forall i \in \mathcal{S}_2} p_i T_i}{\sum_{\forall i \in \mathcal{S}_2} p_i} = \Theta_2.$$

For the second term we have, slightly rearranged:

$$\int t_x \sum_{\forall i \in \mathcal{S}_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i dt_x.$$

The probability of having a stratum $\mathcal{S}_2$ trajectory experiencing its third failure at t_x is the probability that the first (and second) failure has not been dealt with by t_x, i.e. the duration $T_j > t_x, j \in \mathcal{S}_1$ and that a new failure occurs at t_x. These two events are independent. Up to the failure time t_x, the trajectories of strata $\mathcal{S}_1$ and $\mathcal{S}_2$ passing this point are identical. Hence, $\sum_{\forall i \in \mathcal{S}_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i = \Pr\{T_j > t_x\} \pi_1 (n-1)\lambda$ and by partial integration,

$$\int t_x \Pr\{T_j > t_x\} dt_x = \frac{1}{2} E(T_j^2 | j \in \mathcal{S}_1) = \frac{1}{2}(\Theta_1^2 + \sigma_1).$$

Combining the above, inserting it into (15.7), using that $\sum_{\forall i \in \mathcal{S}_2} p_i = \pi_2$ and that due to the small failure intensity $\sum_{k>2} \pi_k \approx \pi_3$, the unconditional probability of a trajectory in stratum $\mathcal{S}_3$ approximately becomes:

$$\begin{aligned} \pi_3 &= (n-2)\lambda(\Theta_2 \pi_2 - \frac{1}{2}(\Theta_1^2 + \sigma_1)\pi_1 (n-1)\lambda) \\ &= (n-1)(n-2)\lambda^2(\Theta_2 \Theta_1 - \frac{1}{2}(\Theta_1^2 + \sigma_1))\pi_1. \end{aligned} \tag{15.8}$$

Since we have that $1 > \pi_1 > 1 - \pi_2 - \pi_3$ and as argued above, a sufficiently accurate estimate for π_1 may be obtained from the lower bound since $1 \approx \pi_1 \approx 1 - \pi_2 - \pi_3$, or slightly more accurately by solving π_i from (15.6), (15.8) and $1 = \pi_1 + \pi_2 + \pi_3$.

15.3.3.2 Estimation Errors

The estimation errors or the uncertainty in the obtained result is computed using the sectioning approach [72]. The experiments are subdivided into $N \sim 10$ independent

runs of the same size. Let $\hat{E}_l(Y)$ be the estimate from the l^{th} of these; then:

$$\hat{E}(Y) = \frac{1}{N}\sum_{l=1}^{N}\hat{E}_l(Y), \quad \hat{\text{Var}}(Y) = \frac{1}{(N-1)}\sum_{l=1}^{N}(\hat{E}_l(Y^2) - \hat{E}^2(Y)).$$

15.4 Experimental Results

This section presents experimental results of fault injections on the target system. A total of 3000 experiments were performed, aiming at 1000 per stratum. Each experiment is classified as being of stratum $\mathcal{S}_k$, if exactly k fault injections occur before the experiment completes (all services are fully recovered). The results of the experiments are presented in Table 15.1. Some experiments "trying to achieve higher order strata" ($\mathcal{S}_3$ and $\mathcal{S}_2$) fall into lower order due to injections being far apart, cf. Figure 15.6(b), or addressing the same node.

Table 15.1: Results obtained from the experiments (in milliseconds).

Classification	Count	$\Theta_k = E(T\|\mathcal{S}_k)$	sd=$\sqrt{\sigma_k}$	Θ_k, 95% conf.int.
Stratum $\mathcal{S}_1$	1781	8461.77	185.64	(8328.98, 8594.56)
Stratum $\mathcal{S}_2$	793	12783.91	1002.22	(12067.01, 13500.80)
Stratum $\mathcal{S}_3$	407	17396.55	924.90	(16734.96, 18058.13)

Of the 3000 experiments performed, 19 (0.63%) were classified as inadequate. In these experiments one or more of the services failed to recover (16 exp.), or they behaved in an otherwise unintended manner. In the latter three experiments, the services did actually recover successfully, but the experiments were classified as inadequate, because an additional (not intended) failure occurred. The inadequate ones are dispersed with respect to experiments seeking to obtain the various strata as follows; two for $\mathcal{S}_1$, 6 for $\mathcal{S}_2$, and 11 for stratum $\mathcal{S}_3$. One experiment resulted in a complete failure of the ARM infrastructure, caused by three fault injections occurring within 4.2 seconds leaving no time for ARM to perform self-recovery. Of the remaining, 13 were due to problems with synchronizing the states between the RM replicas, and 2 were due to problems with the Jgroup membership service. Even though none of the inadequate experiments reached the down state, $D0$, for the MS service, it is likely that additional failures would have caused a transition to $D0$. To be conservative in the

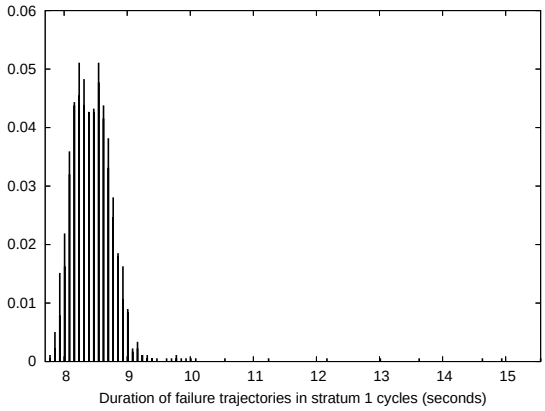
predictions below, all the inadequate experiments are considered to have trajectories visiting down states, and causing a fixed down time of 5 minutes.

Figure 15.7 shows the probability density function (pdf) of the recovery periods for each of the strata. The data for stratum $\mathcal{S}_1$ cycles indicate that it has a small variance. However, 7 experiments have a duration above 10 seconds. These durations are likely due to external influence (CPU/IO starvation) on the machines in the target system. This was confirmed by examining the cron job scheduling times, and the running time of those particular experiments. Similar observations can be identified in stratum $\mathcal{S}_2$ cycles, while it is difficult to identify such observations in $\mathcal{S}_3$ cycles. The pdf for stratum $\mathcal{S}_2$ in Figure 15.7(b) is bimodal, with a peak at approximately 10 seconds and another around 15 seconds. The density of the left-most part is due to experiments with injections that are close, while the right-most part is due to injections that are more than 5-6 seconds apart. The behavior causing this bimodality is due to the combined effect of the delay induced by the view agreement protocol, and a 3 second delay before ARM triggers recovery. Those injections that are close tend to be recovered almost simultaneously. The pdf for stratum $\mathcal{S}_3$ has indications of being multimodal. However, the distinctions are not as clear in this case.

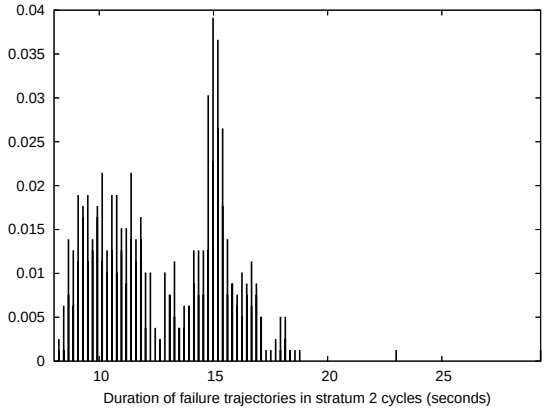
Given the results of the experiments, we are able to compute the expected trajectory durations, Θ_1, Θ_2 and the variance σ_1 as shown in Table 15.1. These are needed to compute the unconditional probabilities π_2 and π_3 given in (15.6) and (15.8) for various node mean time between failures (node MTBF=λ^{-1}), as shown in Table 15.2. The low probabilities of a second and third near-coincident failure is due to the relatively short recovery time (trajectory durations) for strata $\mathcal{S}_1$ and $\mathcal{S}_2$. Table 15.2 compares these values with a typical node recovery (reboot) time of 5 minutes and manual recovery time of 2 hours. These recovery times are computed using fixed values for Θ_1, Θ_2 and the variance σ_1 as shown in the heading of Table 15.2.

Given the unconditional probabilities and the expected down time for each stratum (obtained by measurements), we may use (15.5) and (15.2) to compute estimates for the system unavailability ($\hat{U}$). Similarly, the expected probability of failure for each stratum (obtained by measurements) is used solve (15.5), and consequently estimates for the system MTBF ($\hat{\Lambda}^{-1}$) is obtained by (15.4).

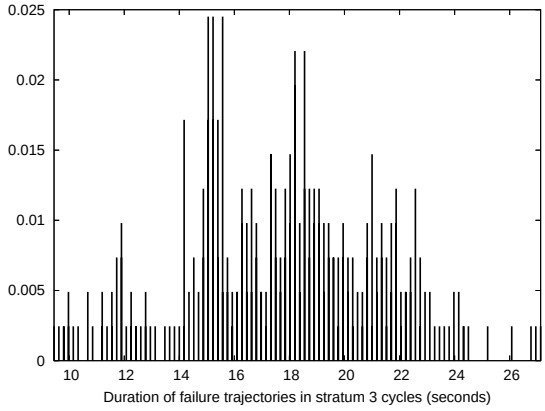
Of the 407 stratum $\mathcal{S}_3$ experiments, only 3 reached a down state. However, we include also the 19 inadequate experiments as reaching a down state. Thus, Table 15.2 provides only indicative results of the unavailability ($\hat{U}$) and MTBF ($\hat{\Lambda}^{-1}$) of the MS service, and hence confidence intervals for these estimates are omitted. The results show as expected, that the two inadequate experiments from stratum $\mathcal{S}_1$ included



(a) Stratum $\mathcal{S}_1$



(b) Stratum $\mathcal{S}_2$



(c) Stratum $\mathcal{S}_3$

Figure 15.7: Probability density functions of trajectory durations for the strata.

Table 15.2: Computed probabilities, unavailability metric and the system MTBF.

	Experiment Recovery Period		Node Recovery (min.) ($\Theta_1 = 5, \Theta_2 = 8, \sqrt{\sigma_1} = 0.5$)		Manual Node Recovery (hrs.) ($\Theta_1 = 2, \Theta_2 = 4, \sqrt{\sigma_1} = 0.5$)	
	Node Mean Time Between Failure (MTBF=λ^{-1}) (in days)					
	100	200	100	200	100	200
π_1	0.99999314	0.99999657	0.99975688	0.99987845	0.99412200	0.99707216
π_2	$6.8556 \cdot 10^{-6}$	$3.4278 \cdot 10^{-6}$	$2.4306 \cdot 10^{-4}$	$1.2153 \cdot 10^{-4}$	$5.8333 \cdot 10^{-3}$	$2.9167 \cdot 10^{-3}$
π_3	$4.0729 \cdot 10^{-11}$	$1.0182 \cdot 10^{-11}$	$5.5953 \cdot 10^{-8}$	$1.3988 \cdot 10^{-8}$	$4.4662 \cdot 10^{-5}$	$1.1165 \cdot 10^{-5}$
$\hat{U}$	$4.6713 \cdot 10^{-7}$	$2.3356 \cdot 10^{-7}$	$2.7771 \cdot 10^{-4}$	$1.3887 \cdot 10^{-4}$	$6.6275 \cdot 10^{-3}$	$3.3236 \cdot 10^{-3}$
$\hat{\Lambda}^{-1}$	20.3367 years	40.6741 years	—	—	—	—

with a service down time of 5 minutes, completely dominates the unavailability of the service. However, accounting for near-coincident failures may still prove important once the remaining deficiencies in the platform have been resolved. Although the results are indicative, it seems that very high availability and MTBF may be obtained for services deployed with Jgroup/ARM.

15.5 Concluding Remarks

This chapter has presented an approach for the estimation of dependability attributes based on the combined use of fault injection and a novel post stratified sampling scheme. The approach has been used to assess and evaluate a service deployed with the Jgroup/ARM framework. The results of the experimental evaluation indicate that services deployed with Jgroup/ARM can obtain very high availability and system MTBF.

The approach may also be extended to provide unbiased estimators, allowing us to determine confidence intervals also for dependability attributes given enough samples visiting the down states.

Chapter 16

Evaluation of Network Instability Tolerance

The recovery performance of Jgroup/ARM has been evaluated experimentally with respect to both node and communication failures. An extensive study of its crash failure behavior is presented in Chapter 15 and also in [57]. Hence, the study in this chapter focuses on the other core feature of Jgroup/ARM, namely its ability to tolerate network instability and partitioning due to network failures. Network instability and partition failures may arise for a number of reasons, e.g. router crashes or power outages, physical link damage, buffer overflows in routers, router configuration errors and so on. The reality of such failures has been confirmed by others through measurements [69, 65].

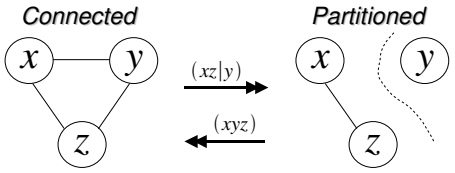
To the author's knowledge, evaluations of network instability tolerance of fault treatment systems has not been conducted before. The Orchestra [35] fault injection tool has been used to evaluate a group membership protocol by discarding selected messages to test the robustness of the protocol. Loki [28] has been used to inject correlated network partitions to evaluate the robustness of the Coda filesystem [71].

At any given time, the connectivity state of the target environment is called the current *reachability pattern*. The reachability pattern may be *connected* (all nodes are in the same partition), or *partitioned* (failures render communication between subsets of nodes impossible). The reachability pattern may change over time, with partitions forming and merging. In our evaluation, reachability patterns are *injected* by the *experiment executor* as discussed in Chapter 13.

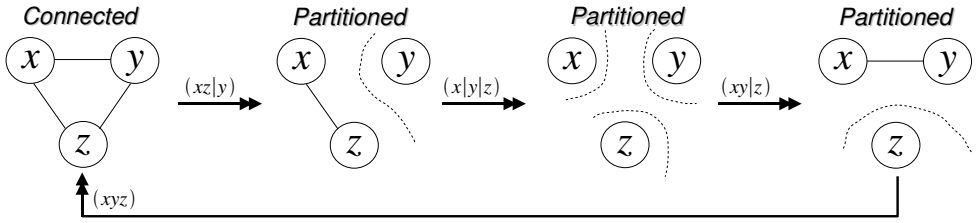
The goal of the experimental evaluation is to test Jgroup/ARM with respect to:

- **Configuration (a)** Its recovery performance when exposed to a single partitioned reachability pattern.
- **Configuration (b)** Its ability to recover when exposed to a rapid succession of different reachability patterns.

A series of experiments have been performed in both configurations. Figure 16.1 illustrates one possible sequence of reachability changes for each of the two configurations; the letters x, y and z refers to the sites in the target environment. Both configurations begin and end in a fully connected network. Configuration (a) injects only a single partition before returning to the original connectivity state, whereas configuration (b) may inject a double partition. Figure 11.6 illustrates the expected ARM behavior for configuration (a) experiments.



(a) Configuration (a).



(b) Configuration (b).

Figure 16.1: Two example sequences of reachability patterns.

In configuration (b) experiments, four reachability patterns are injected at random times, the last one returning to the fully connected reachability pattern. Multiple near-coincident reachability changes may occur before the system stabilizes, i.e. a new reachability pattern may be injected before the previous has been completely handled by ARM.

For the duration of an experiment, events of interest are monitored, and post-experiment analysis is used to construct a single global timeline of events. In the study

analysis density estimates for the various delays involved in detection and recovery are computed.

In the following we present the target system and state machine used for the evaluation. Further, we briefly discuss the experiment execution, and the emulation of reachability patterns. Finally, we present our findings and concluding remarks.

16.1 Target System

Figure 16.2 shows the target system for our measurements. It consists of three sites denoted x, y and z, two located in Stavanger and one in Trondheim (both in Norway), interconnected through the Internet[1].

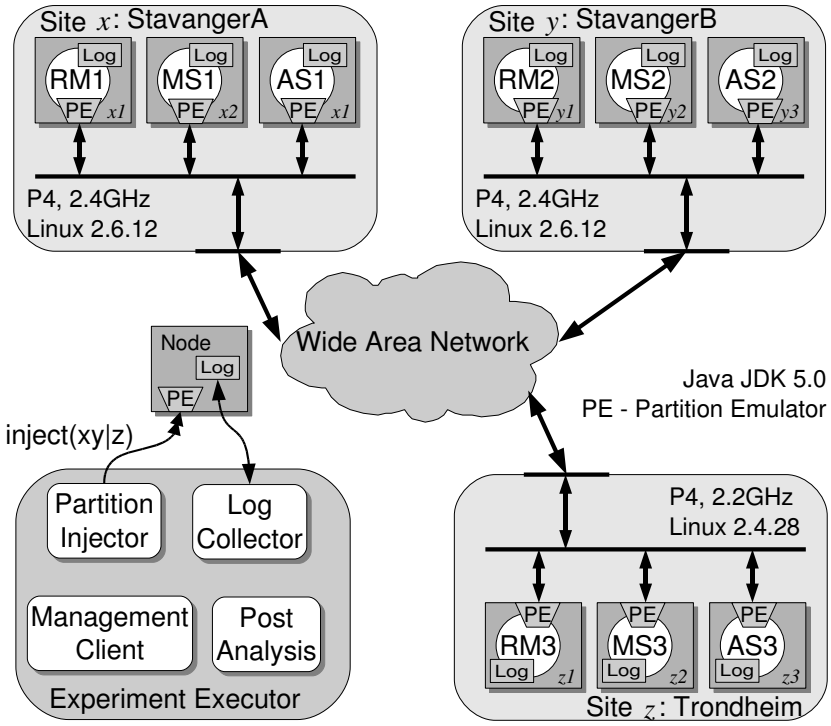


Figure 16.2: Target system used in the experiments.

[1]Originally, Bologna (Italy) were used as the third site, but due to practical problems with clock synchronization this site was replaced with another one in Stavanger instead.

Each site has three nodes denoted $x1, x2, x3$, $y1, y2, y3$ and $z1, z2, z3$. Initially the three nodes with index 1 host the RM, nodes with index 2 host the *monitored service* (MS), while nodes with index 3 host the *additional service* (AS). The latter was added to assess ARM's ability to handle concurrent failure recoveries of different services, and to provide a more realistic scenario. Finally, an external node hosts the *experiment executor*.

The policies used in the experiments are those described in Section 10.1. In all the experiments and for all services, $R_{\text{init}}(*) := 3$ and $R_{\text{min}}(*) := 2$. That is, all services have three replicas initially and ARM tries to maintain at least two replicas of each service in each partition to may arise. The experiments enable simultaneous observations of all services in the target system, including the ARM infrastructure. In the following, however, we report on ARM responsiveness to various network failure scenarios with respect to just our *subsystem of interest*, the MS service.

16.2 A Partial State Machine and Notation

In an attempt to model the failure-recovery behavior of the MS service under various reachability patterns, a global state machine representation is defined. The state machine is used to perform sanity checks, and to identify sampling points for our measurements. However, due to the large number of states, only the initial states are shown in Figure 16.3 and a trace snapshot in Figure 16.4. Note that since each site in this study has at least one (assumed non-crashing) replica of each service, there are no *down* states.

Each state is identified by its global reachability pattern, the number of replicas in each partition and the number of members in the various (possibly concurrent) views. The number of replicas in a partition is the number of letters x, y, and z that are not separated by a | symbol. The different letters refer to the site in which a replica resides. The | symbol indicates a disconnection between the replicas on its left- and right-hand side. The number in parenthesis in each partition is the number of members in the view of that partition. A partition may for short periods of time have multiple concurrent views, as indicated by the + symbol. Concurrent views in the same partition are not stable, and a new view including all live members in the partition will be installed, unless interrupted by a new reachability change. Two examples: The fully connected steady state is identified by $[xyz(3)]$ in which each site has a single replica, and all have installed a three member view. In the state

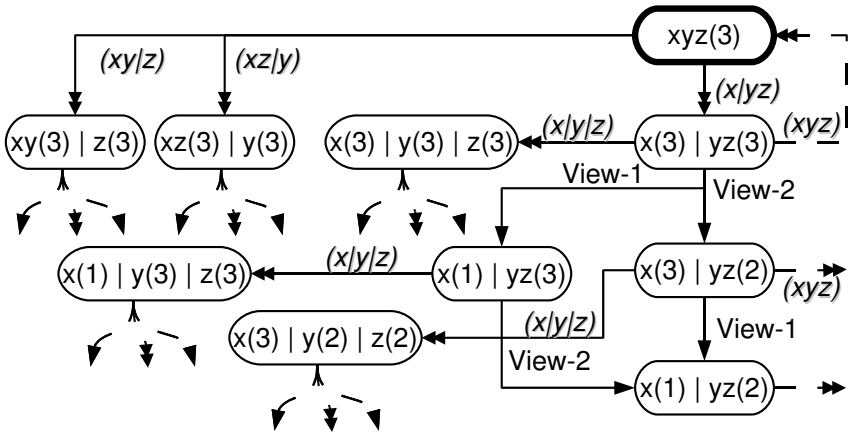


Figure 16.3: Excerpt of the state machine showing the initial states/transitions.

$[xx(2) \mid yy(1+1) \mid zz(2)]$ all sites are disconnected from each other and all have installed an extra replica to satisfy the $R_{\text{min}} = 2$ requirement. However, the replicas in site y still have not installed a common two-member view.

Each state can be classified as stable (bold outline in Figure 16.3 and Figure 16.4) or unstable. Let R_p denote the current redundancy level of a service in partition p. A state is considered *stable* if

$$(R_p = |\mathcal{V}_p| \wedge R_{\text{init}} \geq R_p \geq R_{\text{min}}) \qquad \forall p \in \mathcal{P}$$

where $\mathcal{P}$ is the current set of partitions. In the stable states, no ARM action is needed to increase or decrease the service redundancy level. All other states are considered *unstable*, meaning that more events are needed to reach a stable state. Once in a stable state, only a new reachability change can cause it to enter into an unstable state.

In Figures 16.3 and 16.4 we distinguish between *system events* (regular arrow) and *reachability change events* (double arrow). In the evaluation, only relevant events are considered such as *view change*, *replica create*, *replica remove* and *reachability change* events. View-c denote a view change, where c is the cardinality of the view. Reachability change events are denoted by (xyz), where a $|$ symbol indicate which sites are to be disconnected. A new reachability change may occur in any state; note in particular the transition from the unstable state $[x(1) \mid yz(2)]$ in Figure 16.4, illustrating that recovery in partition x did not begin before a new reachability change arrived.

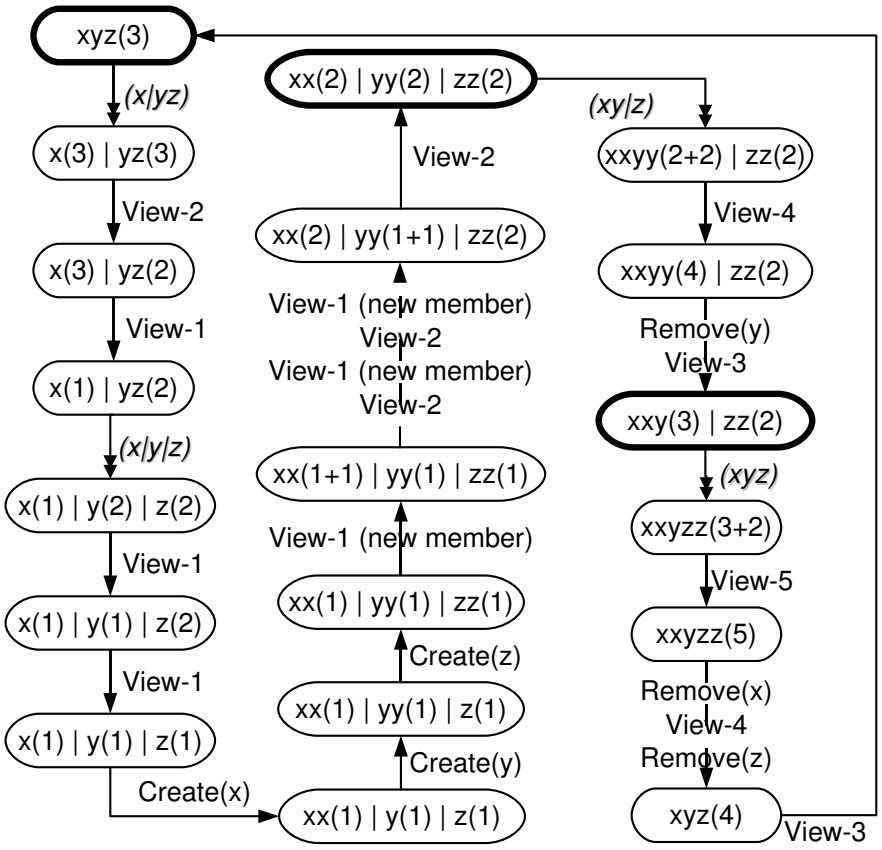


Figure 16.4: An example state machine trace. The dashed arrows with multiple events are used to reduce the size of the figure.

16.3 Measurements

This section presents the evaluation of the two configurations in this study. In configuration (a) two reachability changes are scheduled to be injected in the target system, whereas in configuration (b) four injections are scheduled. Both the generation of injection times and the selection of the reachability patterns follow a uniform distribution. Each experiment begins and ends in the fully connected steady state [$xyz(3)$], while the intermediate reachability patterns may cause a single or double network partition.

Figure 16.5 shows a timeline of injection events. Let I_i denote the injection time of the i^{th} injection event, and let E_i be the time of the last system event before

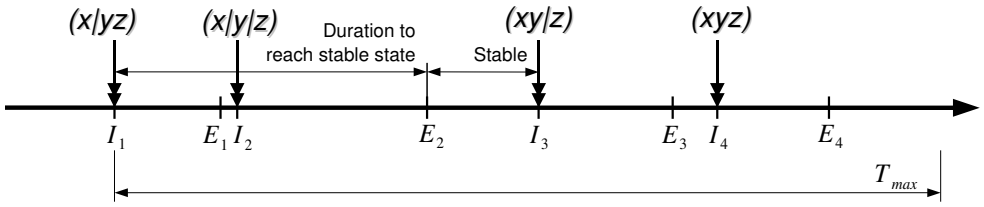


Figure 16.5: Sample timeline with injection events.

I_{i+1}. E_i events will bring the system into either a stable or unstable state. Nearly-coincident injections tend to bring the system to an unstable state, while injections spaced further apart typically enable the recovery to a stable state. Let D_i denote the duration needed to reach a stable state after injection I_i, where D_i may extend beyond I_j, where $j > i$, before reaching a stable state. Figure 16.6 illustrates the duration D_2 for three different injection scenarios that may occur in configuration (b) experiments.

The behavior of Jgroup/ARM in response to such injections is observed, and a global event trace is computed from each experiment, allowing us to compute the trajectory of visited states and the time spent in each of the states. Consequently, this allows us to extract various detection and recovery delays and to determine the correctness of trajectories.

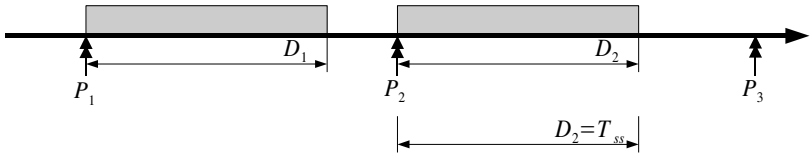
16.3.1 Injection Scheme

The following injection scheme is applied to emulate real reachability patterns. Let $P_i^{(a)}$ be the set of reachability patterns from which injections for configuration (a) are chosen:

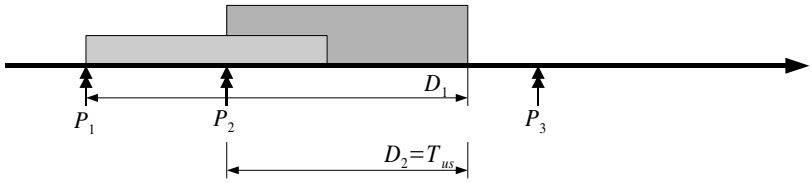
$$P_i^{(a)} = \begin{cases} \{(xy|z), (x|yz), (xz|y)\} & i = 1 \\ \{(xyz)\} & i = 0, 2 \end{cases}$$

where i denotes the injection number and $i = 0$ is the initial state. Similarly, let P_i be the set of patterns from which injections for configuration (b) are chosen:

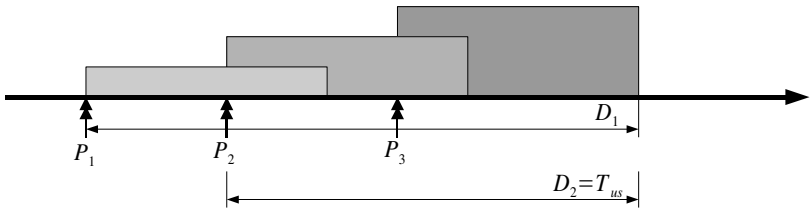
$$P_i = \begin{cases} \{(xy|z), (x|yz), (xz|y)\} & i = 1, 3 \\ \{(xyz), (x|y|z)\} & i = 2 \\ \{(xyz)\} & i = 0, 4 \end{cases} \quad (16.1)$$



(a) Injection of a P_2 pattern starting from a stable state.



(b) Injection of a P_2 pattern starting from an unstable state; aborting recovery activation for the P_1 pattern.



(c) Injection of a P_2 pattern starting from an unstable state; recovery is interrupted by another reachability change, P_3, before stabilizing.

Figure 16.6: Timelines illustrating the injection of a P_2 pattern. Shaded regions indicate periods where the system is in unstable states.

Then let $p_{j,i}$ denote the i^{th} reachability pattern to inject in the j^{th} experiment, where $p_{j,i}$ is drawn uniformly and independently from P_i. The patterns in P_i are organized in this manner to closely model reachability patterns that could occur in real disconnection scenarios, cf. Figure 16.1.

Each injection time I_i is uniformly distributed over the interval $[T_{\text{min}}, T_{\text{min}}+T_{\text{max}}\rangle$, where T_{min} is the minimal distance between two injections. In each configuration, T_{max} is chosen to be longer than any foreseen trajectory to reach a stable state. This choice is motivated by the intent to test failure recovery for injections occurring over the whole interval of unstable states. For configuration (a), $T_{\text{min}} = 15$ seconds is used to ensure that the final injection (xyz) occurs after the system has reached a stable state, while $T_{\text{min}} = 0$ is used for configuration (b).

The sampling scheme for configuration (b) includes the following steps:

1. The first reachability pattern, $i = 1$, is set to occur at $I_1 = 0$. The following 3 reachability change instants are drawn uniformly distributed over the interval $[0, T_{\text{max}}\rangle$ and sorted such that $I_i \leq I_{i+1}$ yielding the ordered set $\{I_1, I_2, I_3, I_4\}$. Let k be the index denoting the number of reachability changes injected so far. Initially, $k := 1$.
2. While $k \leq 4$ do
   (a) Inject the k^{th} reachability change, drawn from P_k, at time I_k.
   (b) Set $k := k + 1$

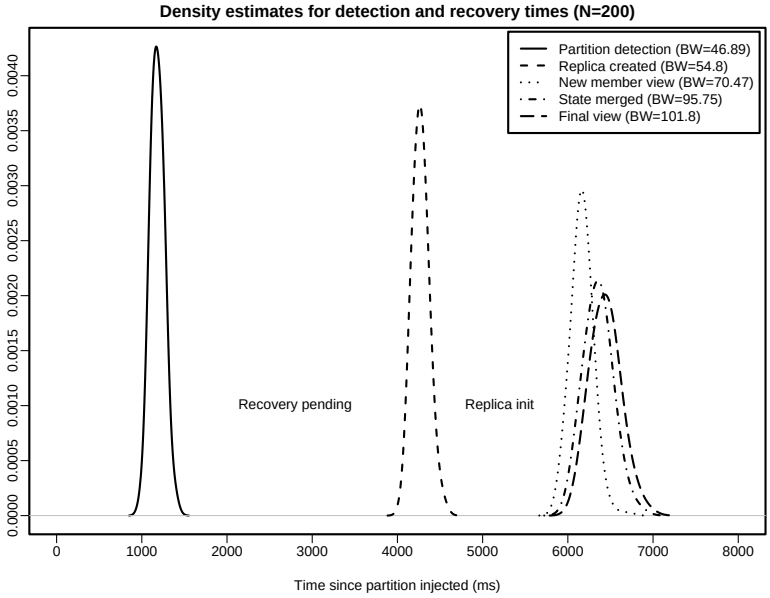
16.4 Experimental Results

This section presents the experimental results of our reachability injections. In 99.6% of the experiments the expected final state $[xyz(3)]$ was reached, i.e. the initial redundancy level was restored after the final (xyz) injection. Six experiments (out of 1500) failed to reach the final state.

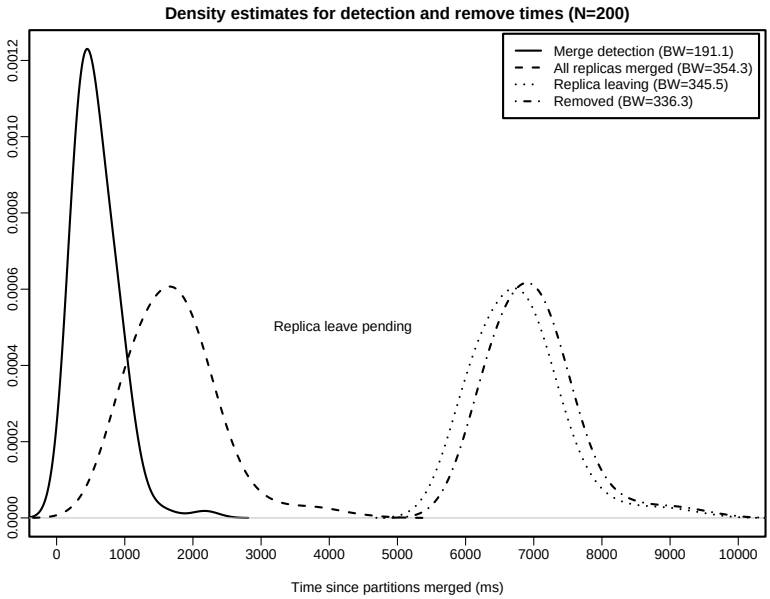
In the following, various measures are obtained for the system's behavior between the first and last injection. The results are presented in the form of kernel density estimates, which gives an estimate of the probability density function; for details see [72]. The smoothing bandwidth (BW), given in the legend of each curve, is chosen as a trade-off between detail and smoothness.

16.4.1 Configuration (a)

In configuration (a), a total of 200 experiments were performed and density estimates obtained using a Gaussian kernel [72]. In this configuration, all experiments completed successfully, reaching the final state $[xyz(3)]$. Figure 16.7(a) shows the density estimates for the time of the various events in the recovery cycle after the injection of a network partition drawn from $P_1^{(a)}$, e.g. $(xy|z)$. All measurements in the plots are relative to the injection time, and appear to follow a normal distribution. The *partition detection* curve is the time it takes to detect that a partition has occurred; that is, the time until the member in the "single site partition" installs a new view. It is this view that triggers recovery in that partition to restore the redundancy level back to R_{min} =2. The *recovery pending* period is due to the 3 second safety margin (`ServiceMonitor expiration`) ARM uses to avoid triggering



(a) Network partition delays.



(b) Network merge delays.

Figure 16.7: Density estimates for the various delays in configuration (a) experiments.

unnecessary recovery actions, cf. Line 18 in Listing 10.2. The *replica init* period is the time it takes to create a new replica to compensate for lack of redundancy. This period is mostly due to JVM initialization, including class loading. The *new member view* curve is the first singleton view installed by the new replica; this occurs as soon as the replica has been initialized. The *state merge* curve is the start of the state transfer from the single remaining replica to the new replica, while the *final view* curve marks the end of the state merge after which the system is stable.

Figure 16.7(b) shows the time of events in the remove cycle after the injection of the final (xyz) merge pattern. *Merge detection* is the time until the first member installs a new view after the merge injection. This first view is typically not a full view containing all four members, whereas the *all replicas merged* curve is the time it takes for all members to form a common view. The tail of this curve is due to delays imposed by the membership service when having to execute several runs of the view agreement protocol before reaching the final four member view. The *replica leave pending* period is due to the 5 second `RemoveDelay` used by the supervision module to trigger a leave event for some member (see Section 11.3.3). The last two curves indicate the start of the leave request and installation of a three member view which brings the system back to the steady state $[xyz(3)]$.

16.4.2 Configuration (b)

For configuration (b), 1500 experiments were performed and density estimates obtained using the Epanechnikov kernel [72]. In this configuration, six experiments (0.4%) failed to reach the final state $[xyz(3)]$ due to a problem with the view agreement protocol stopping after the final merge pattern was injected. This problem seems to occur in rare circumstances when reachability changes arrive in rapid succession. Unfortunately, not enough debug information was logged during the experiments to accurately diagnose the problem. However, there are indications that the problem is related to a failure to retransmit messages in the Jgroup multicast layer. This can happen if a message is believed to have been received by all servers in the destination set, and the message is consequently discarded from the sender's buffer, after which it cannot be retransmitted. The multicast layer requires that a message sent by a correct server is eventually received by all servers in the destination set [87]; retransmission is the mechanism used to ensure this. In rare cases it seems that a bug can cause messages to be discarded prematurely. However, the exact cause of the problem needs further analysis. In the remaining discussion, the six failed experiments are not considered.

Table 16.1 shows the number of observations for the different combinations of reachability changes injected during configuration (b) experiments. The table shows how many occurrences were observed for a particular reachability change given that the injection was performed when in a stable or unstable state.

In each experiment there are four injections as shown in Figure 16.5 (and also in Figure 16.1(b)), each followed by series of system events or another injection. The i^{th} reachability pattern to be injected is drawn from the set P_i in (16.1).

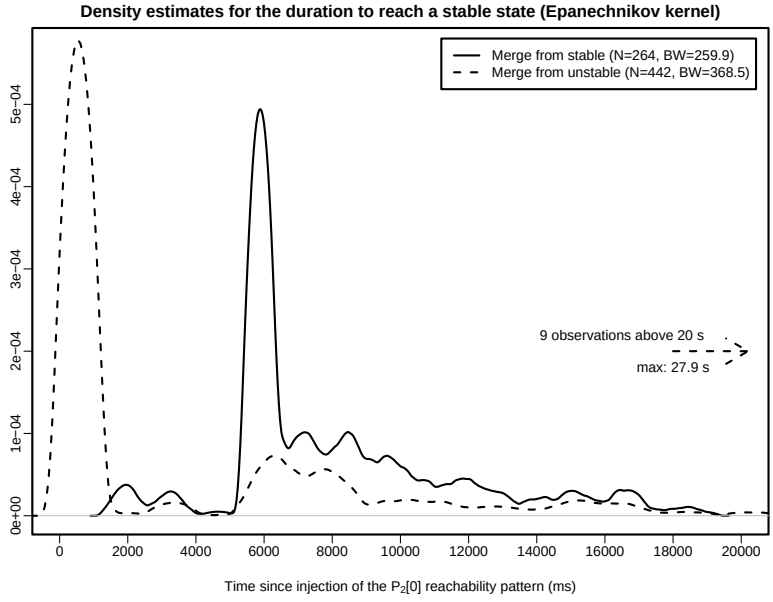
Table 16.1: Number of observations for different combinations of injections starting from a stable or unstable state.

			Starting from			
	Injection	Reachability change	Unstable	Stable		Aggregate
1	I_1	$P_0 \rightarrow P_1$	—	1494		1494
2	I_2	$P_1 \rightarrow P_2[0]$	442	264	706	1494
3	I_2	$P_1 \rightarrow P_2[1]$	497	291	788	
4	I_3	$P_2[0] \rightarrow P_3$	276	430	706	1494
5	I_3	$P_2[1] \rightarrow P_3$	500	288	788	
6	I_4	$P_3 \rightarrow P_4$	844	650		1494

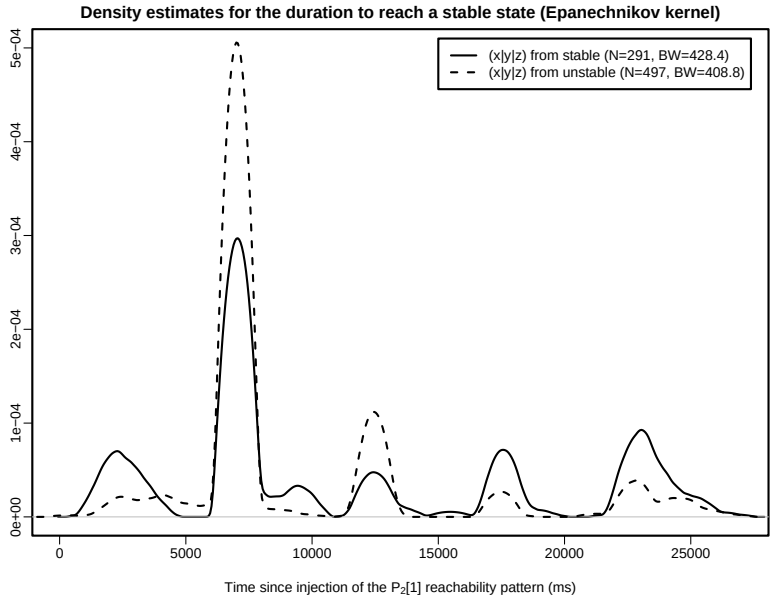
The plot in Figure 16.8(a) shows density estimates for D_2 when starting from a (1) stable or (2) unstable P_1 reachability pattern and entering a $P_2[0] = (xyz)$ pattern (line 2 in Table 16.1), i.e. a fully connected network.

In case (1) (solid curve), when starting from a stable P_1 pattern (cf. Figure 16.6(a)) the following observations have been made:

- The peak at about 6 s (approximately 119 observations) is due to removal of an excessive replica installed while in the P_1 pattern. This behavior corresponds to the *removed* curve (which corresponds to a stable state) in Figure 16.7(b).
- The 17 observations before the peak are due to experiments that briefly visits $P_2[0]$ before entering a P_3 pattern equal to the initial P_1 pattern. These observations do not trigger removal since they are caused by rapid reachability changes.
- The rather long tail after the 6 s peak is due to variations of the following scenario: In the $P_2[0]$ pattern a remove is triggered and before stabilizing a P_3 pattern is injected. Due to the remove, there is again a lack of redundancy, thus ARM triggers another recovery action. Some experiments stabilize in P_3,



(a) Duration to reach a stable state after a $P_2[0]$ injection.



(b) Duration to reach a stable state after a $P_2[1]$ injection.

Figure 16.8: Density estimates for the duration D_2 to reach a stable state after a P_2 injection in configuration (b) experiments.

while others do not complete until reaching the final $[xyz(3)]$ state in P_4. This depends on the time between reachability changes.

For case (2) (dashed curve), starting from an unstable P_1 pattern the following observations have been made:

- There is a peak at about 0.6 s due to injections that only briefly visits the P_1 pattern, quickly reverting the partition to a $P_2[0] = P_0$ pattern, taking the system back to the $[xyz(3)]$ state without triggering any ARM actions. This can happen if $P_2[0]$ occurs before the 3 s safety margin expires. A total of 288 observations constitute this peak; 54 of these are due to two consecutive injections without intermediate system events. Figure 16.6(b) illustrates this scenario.
- There are also 7 observations below 6 s that are due to ARM triggering recovery in the P_1 reachability pattern (recovery is not completed in P_1, hence unstable) and is then interrupted by a short visit to the $P_2[0]$ pattern before entering a P_3 pattern identical to the initial P_1 pattern. Figure 16.6(c) illustrates this scenario.
- The observations above 6 s are similar to the above, except that recovery is completed in $P_2[0]$ leading to a $[xyzz(4)]$ (or similar) state. Consequently, removal of the excessive replica is triggered in $P_2[0]$, but not completed. Hence, it needs to enter P_3 before reaching stable. It may also need to enter P_4 depending on the time between the reachability changes. The variation seen for these observations are caused by the varying time between reachability changes.
- There are 37 observations above 13 s. These follow the scenario above, but the P_3 pattern selected is different from P_1, causing the need to install another replica in the new single site partition. Note that the other partition will then have three replicas. That is, it may stabilize in P_3 in a state similar to $[xxy(3v)|zz(2v)]$, or in P_4 in the $[xyz(3)]$ state. In latter case, two removes are needed before reaching a stable state, since there will be five replicas in the merged partition. For case (2) this is the behavior that gives the longest durations for D_2.

The plot in Figure 16.8(b) shows density estimates for D_2 when starting from a (3) stable or (4) unstable P_1 reachability pattern and entering a $P_2[1] = (x|y|z)$ pattern (line 3 in Table 16.1), i.e. a double partition.

For both case (3) (solid curve) and case (4) (dashed curve), there are multiple peaks at approximately the same time intervals. The observations for case (4) are mostly

due to the same behaviors as in case (3), except that the initial P_1 pattern has not reached a stable state before the $P_2[1]$ reachability change. Hence, we focus only on explaining case (3):

- The small peak at approximately 2.5 s (44 observations) is due to short visits to the $P_2[1]$ pattern without triggering recovery before entering a P_3 pattern equal to the P_1 pattern.
- The main peak at 7 s (117 observations) is due to recovery in the two new single site partitions eventually leading to a $[xx(2)|yy(2)|zz(2)]$ stable state. Recall that the initial P_1 pattern is in a stable state similar to $[xx(2)|yz(2)]$ before the double partition injection. This peak is roughly comparable to the *final view* curve in Figure 16.7(a).
- The small peak just below 10 s (15 observations) is due to short visits to $P_2[1]$ and P_3 before stabilizing in P_4 having to remove one replica created in P_1.
- The peak at 12.5 s (24 observations) is due to recovery initiated in $P_2[1]$, interrupted by a P_3 injection, cf. Figure 16.6(c). Recovery is completed in P_3 followed by a replica remove event bringing the system back to a stable state.
- The peak at 17.5 s (31 observations) is due to brief visits to $P_2[1]$ followed by a P_3 pattern different from P_1 triggering recovery, which eventually completes in P_4 by removing two excessive replicas before reaching stable. Recall that two removals are separated by the 5 s `RemoveDelay`.
- The peak at 23.5 s (60 observations) is due to recovery initiated in $P_2[1]$, which does not stabilize until reaching P_4. Hence, three removals are needed in this case, each separated by 5 s.

16.5 Concluding Remarks

The results obtained from our experiments show that Jgroup/ARM is robust with respect to failure recovery, even in the presence of multiple near-coincident reachability changes. Only six experiments (0.4%) failed to reach the expected final state $[xyz(3)]$ when exposed to frequent reachability changes. Further analysis is needed to fully understand the cause of the problem and to be able solve it.

The delays observed in the measurements are mainly due to two components: (i) execution of protocols, e.g. the view agreement protocol, and (ii) timers to avoid activating fault treatment actions prematurely. Premature activation could potentially

occur when the system is heavily loaded or is experiencing high packet loss rates. The timers constitutes the majority of the observed delays.

Part V

Conclusions

Chapter 17

Conclusions and Further Work

The recent call for papers for the second workshop on *Hot Topics in System Dependability* (HotDep '06) recognizes the relevance of the main contributions of this dissertation by the following (partial) list of requested topics [59]:

- Automated failure management
- Techniques for detection, diagnosis, and recovery from failures
- Metrics and techniques for quantifying dependability

These are all topics that have been studied in this dissertation.

Main Contributions

In this thesis, we have presented the architectural design, implementation and an extensive experimental evaluation of Jgroup/ARM. Jgroup is an object group system that extends the Java distributed object model. ARM is an autonomous replication management framework that extends Jgroup and enables the automatic deployment of replicated objects in a distributed system according to application-dependent policies.

The primary goal of Jgroup/ARM has been to support reliable and highly-available application development and deployment in partitionable systems. Jgroup/ARM addresses an important requirement for modern applications that are to be deployed in networks where partitions can be frequent and long lasting. ARM is a self-managing fault treatment framework that augments Jgroup, enabling simplified administration

of replicated services, made possible through configurable distribution and replication policies. As our experimental evaluations demonstrate, Jgroup/ARM is very robust in its handling of failures, even when exposed to multiple nearly-coincident crash failures or reachability changes. The results of the experiments show that ARM is able to detect failures quickly and to recover from them. Most experiments successfully recover to the expected steady state when exposed to failures.

The thesis also presents a novel approach to the estimation of dependability attributes based on the combined use of randomized crash fault injection and a post stratified sampling scheme. This approach has been used to assess and evaluate a service deployed with the Jgroup/ARM framework. The results of the experimental evaluation indicate that services deployed with Jgroup/ARM can obtain very high availability and system MTBF.

Furthermore, an advanced approach to performing experimental evaluation of the network instability tolerance of Jgroup/ARM has been developed, based on randomized reachability change injections.

The experiment framework used to conduct the experiments in this thesis has proven to be exceptionally useful by uncovering at least a dozen subtle bugs, allowing systematic stress and regression testing.

Directions for Future Research

During the course of this research new insights into potential improvements for the Jgroup/ARM middleware or similar systems have been gained. There are many challenges still ahead and below we summarize some of the open issues and give ideas for further extensions of Jgroup/ARM.

Decentralized recovery management In the current framework the remove policy is implemented through a decentralized mechanism for removing excessive replicas, whereas replica distribution and recovery management is performed by a centralized entity, the replication manager. Further study is needed, but it may also be possible to perform replica recovery through a decentralized mechanism, eliminating the need for a centralized replication manager. The groups themselves could manage the recovery process.

Cold standby for recovery A future extension to the ARM recovery mechanism could be to proactively install cold standby replicas, which do not join the group until

found necessary by ARM. Furthermore, excessive replicas could just leave the group rather than to terminate themselves, thus remaining in a ”cold“ standby state. This extension would avoid the JVM initialization delay of starting a new replica, hence shortening the recovery delay.

Generalize the experimental toolbox In the area of testing and validating distributed systems through fault injection there are still many open issues; in particular on the study analysis part. Such analysis often requires specific knowledge of the underlying system, however, generalizing portions of this analysis may still be possible. Also, a generic toolbox for experimental evaluation could possibly be integrated with existing project management tools, e.g. Maven [75].

Redesign the communication layers The various communication layers in Jgroup are partially based on IP multicast, UDP/IP and RMI. This mixture of communication mechanisms, and the lack of clear separation between the various layers has proven to be very difficult to maintain. In addition, significant performance gains can be obtained by using new technologies such as the new IO framework of Java, `java.nio`, and taking care to avoid internal message copying.

Reliable communication based on IPv6 Taking advantage of anycast and the advanced multicast capabilities of IPv6 in wide area networking to develop a reliable multicast layer is an interesting topic for future research.

Improved separation between policy and framework Further study is needed to simplify user specified policies and to support conflict resolution between policies. This could possibly utilize the PMAC [2] policy management framework from IBM.

Tighter integration between upgrade and recovery management This is necessary to avoid conflicting policy specifications for upgrade and recovery. Furthermore, implementation of the new upgrade approach, proposed in Section 12.5, is likely to reduce the performance overhead of upgrading replicated services.

Tighter integration between the replication manager and dependable registry There is already a dependency between the RM and the DR, and as discussed in Chapter 11 they are co-located for this reason. A tighter integration between these components could be exploited to improve the efficiency of the overall system, since it would only need to keep one database of object groups up-to-date. The drawback however, is reduced independence and modularity between the DR and RM components, as the DR could not be used without the RM and hence recovery would be enabled by default.

More advanced experiment configurations So far experiments have mainly been performed without introducing clients generating system load. Such experiments would give new insight into the performance that can be obtained using Jgroup/ARM. Furthermore it may also enable us to predict the availability perceived by clients.

Integration with Enterprise Java Beans Communication in the EJB framework is based on Java RMI. Jgroup was designed to support the same semantics as RMI. Hence, a future extension could be to enhance EJB with group communication support based on Jgroup.

Group communication and recovery support in MANETs In the future, mobile ad hoc networks, or MANETs, may give rise to new application areas needing group communication and recovery support. Wireless networks have completely different requirements from fixed line networks, e.g. due to a continuously changing topology and battery limitations of mobile nodes, demanding that message transmission be reduced to a bare minimum. Only recently have people started to investigate protocols to support group communication in MANETs [112]. Given a group communication system for MANETs, ARM-like functionality may be implemented on top of it to support fault treatment.

Autonomic management of network services DHCP, DNS and NTP are essential network services that are needed for correct behavior of the network. An improvement in system availability can be expected if such network services were to be implemented on top of an autonomic management framework similar to Jgroup/ARM.

Bibliography

[1] Finn Arve Aagesen, Bjarne E. Helvik, Vilas Wuwongse, Hein Meling, Rolv Bræk, and Ulrik Johansen. Towards a Plug and Play Architecture for Telecommunications. In Thongchai Yongchareon, Finn Arve Aagesen, and Vilas Wuwongse, editors, *Proceedings of the IFIP TC6 Fifth International Conference on Intelligence in Networks (SmartNet)*, pages 321–334, Pathumthani, Thailand, November 1999. Kluwer Academic Publishers. pages 6

[2] Dakshi Agrawal, Kang-Won Lee, and Jorge Lobo. Policy-Based Management of Networked Computing Systems. *IEEE Communications Magazine*, 43(10):69–75, October 2005. pages 137, 138, 235

[3] Yair Amir, Claudiu Danilov, and Jonathan Stanton. A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, New York, June 2000. pages 5, 35, 36, 55, 67, 94, 112, 195

[4] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins, and David Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, February 1990. pages 174, 196

[5] Jean Arlat, Martine Aguera, Yves Crouzet, Jean-Charles Fabre, Eliane Martins, and David Powell. Experimental Evaluation of the Fault Tolerance of an Atomic Multicast System. *IEEE Transactions on Reliability*, 39(4):455–467, October 1990. pages 174, 196

[6] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, fourth edition, 2005. pages 24, 34, 50, 82, 89, 90, 102, 103

[7] Ken Arnold, Bryan O'Sullivan, Jim Waldo, Ann Wollrath, and Robert Scheifler. *The Jini Specification*. Addison-Wesley, second edition, 2001. pages 4, 18, 23, 36, 45, 130

[8] Autonomic Communication. http://www.autonomic-communication.org/. Last visited May 2006. pages 6

[9] IBM, Autonomic Computing. http://www.research.ibm.com/autonomic/. Last visited May 2006. pages 6

[10] Availigent, Duration software. http://www.availigent.com/. Last visited May 2006. pages 4, 20

[11] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January-March 2004. pages 15

[12] Dimiter Avresky, Jean Arlat, Jean-Claude Laprie, and Yves Crouzet. Fault Injection for Formal Testing of Fault Tolerance. *IEEE Transactions on Reliability*, 45(3):443–455, September 1996. pages 197

[13] Özalp Babaoğlu, Alberto Bartoli, and Gianluca Dini. Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997. pages 51

[14] Özalp Babaoğlu, Renzo Davoli, and Alberto Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, April 2001. pages 16, 43, 46, 195

[15] Özalp Babaoğlu, Renzo Davoli, Alberto Montresor, and Roberto Segala. System Support for Partition-Aware Network Applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 184–191, Amsterdam, The Netherlands, May 1998. pages 28, 52

[16] Özalp Babaoğlu, Hein Meling, and Alberto Montresor. Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002. pages 7

[17] Özalp Babaoğlu and André Schiper. On Group Communication in Large-Scale Distributed Systems. In *Proceedings of the ACM SIGOPS European Workshop*, pages 612–621, Dagstuhl, Germany, September 1994. Also appears as ACM SIGOPS Operating Systems Review, 29 (1):62–67, January 1995. pages 28, 45

[18] Bela Ban. JavaGroups – Group Communication Patterns in Java. Technical report, Department of Computer Science, Cornell University, July 1998. pages 4, 5, 34, 35, 55, 74, 76, 94

[19] Arash Baratloo, P. Emerald Chung, Yennun Huang, Sampath Rangarajan, and Shalini Yajnik. Filterfresh: Hot Replication of Java RMI Server Objects. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico, April 1998. pages 34, 35, 36

[20] P. A. Barrett, A. M. Hilborne, P. G. Bond, D. T. Seaton, P. Verissimo, L. Rodriguez, and N. A. Speirs. The Delta-4 Extra Performance Architecture (XPA). In Brian Randell, editor, *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS '90)*, pages 481–489, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society Press. pages 31

[21] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993. pages 26, 28, 47

[22] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications and Prentice Hall, December 1996. pages 13

[23] Kenneth P. Birman and Thomas A. Joseph. Exploiting Virtual Synchrony in Distibuted Systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 123–138, 1987. pages 33, 107

[24] The BISON project. http://www.cs.unibo.it/bison/. Last visited May 2006. pages 6

[25] Andrea Bondavalli, Silvano Chiaradonna, Domenico Cotroneo, and Luigi Romano. Effective Fault Treatment for Improving the Dependability of COTS and Legacy-Based Applications. 1(4):223–237, 2004. pages 37

[26] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The Primary-Backup Approach. In Sape Mullender, editor, *Distributed Systems*,

chapter 8, pages 199–216. Addison-Wesley, second edition, 1994. pages 31, 104

[27] Salvatore Cammarata. Studio e implementazione di un protocollo di reliable multicast e failure detector per servizi di group communication. Master's thesis, Department of Computer Science, University of Bologna, 2000. In Italian. pages 114, 119, 121

[28] Ramesh Chandra, Ryan M. Lefever, Kaustubh R. Joshi, Michel Cukier, and William H. Sanders. A Global-State-Triggered Fault Injector for Distributed System Evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605, July 2004. pages 174, 175, 196, 198, 215

[29] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):1–43, December 2001. pages 5, 26, 27

[30] P.E. Chung, Y. Huang, S. Yajnik, D. Liang, and J. Shih. DOORS: Providing Fault-Tolerance for CORBA Applications. In *Proceedings of the IFIP International Conference on Distributed System Platforms and Open Distributed Processing (Middleware '98)*, September 1998. pages 34, 36, 37

[31] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *eclipse AspectJ*. Addison-Wesley, 2004. pages 184

[32] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems — Concepts and Design*. Addison-Wesley, fourth edition, 2005. pages 107, 119, 129, 150

[33] Geoff Coulson, J. Smalley, and Gordon S. Blair. The Design and Implementation of a Group Invocation Facility in ANSA. Technical Report MPG-92-34, Distributed Multimedia Research Group, Department of Computing, Lancaster University, UK, 1992. pages 27, 41

[34] Michel Cukier, David Powell, and Jean Arlat. Coverage Estimation Methods for Stratified Fault-Injection. *IEEE Transactions on Computers*, 48(7):707–723, July 1999. pages 196, 205

[35] Scott Dawson, Farnam Jahanian, and Todd Mitton. ORCHESTRA: A Fault Injection Environment for Distributed Systems. Technical Report CSE-TR-318-96, University of Michigan, EECS Department, 1996. pages 174, 215

[36] Xavier Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, August 2000. Number 2229. pages 31, 32, 107, 108

[37] Xavier Défago, André Schiper, and Nicole Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998. pages 32

[38] Danny Dolev and Dalia Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4), April 1996. pages 26, 28

[39] Microsoft .NET. http://www.microsoft.com/net/. Last visited May 2006. pages 4, 18

[40] Peder Emstad, Poul E. Heegaard, and Bjarne E. Helvik. *Dependability and Performance in Information and Communication Systems; Fundamentals*. Department of Telematics, NTNU/Tapir, August 2002. pages 205

[41] Pascal Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, January 1998. Number 1867. pages 20, 26, 27, 29, 33, 34, 35, 94, 195

[42] Pascal Felber, Xavier Défago, Patrick Eugster, and André Schiper. Replicating CORBA objects: a marriage between active and passive replication. In *Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, Helsinki, Finland, June 1999. pages 32, 94

[43] Pascal Felber, Rachid Guerraoui, and André Schiper. The Implementation of a CORBA Object Group Service. *Theory and Practice of Object Systems*, 4(2):93–105, January 1998. pages 4, 5, 34, 35, 36, 55

[44] Pascal Felber, Ben Jai, Mark Smith, and Rajeev Rastogi. Using semantic knowledge of distributed objects to increase reliability and availability. In *Proceedings of the 6th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 153–160, Rome, Italy, January 2001. pages 95

[45] Pascal Felber and Priya Narasimhan. Experiences, Approaches and Challenges in Building Fault-Tolerant CORBA Systems. *IEEE Transactions on Computers*, 53(5):497–511, May 2004. pages 20

[46] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, 1983. pages 94

[47] Groovy. http://groovy.codehaus.org/. Last visited May 2006. pages 177

[48] Object Management Group. Fault Tolerant CORBA Using Entity Redundancy. OMG Request for Proposal orbos/98-04-01, Object Management Group, Framingham, MA, April 1998. pages 5

[49] Object Management Group. Fault Tolerant CORBA Specification. OMG Technical Committee Document ptc/00-04-04, Object Management Group, Framingham, MA, April 2000. pages 4, 19, 34, 36

[50] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Rev. 3.0*. Object Management Group, Framingham, MA, June 2002. pages 4, 18, 19

[51] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997. pages 31, 104

[52] Ulf Gunneflo, Johan Karlsson, and Jan Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 340–347, Chicago, IL, USA, June 1989. pages 174, 196

[53] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A Formal Framework for On-line Software Version Change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996. pages 162

[54] Vassos Hadzilacos and Sam Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5. Addison-Wesley, second edition, 1993. pages 33

[55] Mark Hayden. *The Ensenble System*. PhD thesis, Department of Computer Science, Cornell University, January 1998. pages 74, 76

[56] Bjarne E. Helvik. Dependable computing systems and communication networks, January 2001. Draft preprint. pages 14, 16, 29, 31

[57] Bjarne E. Helvik, Hein Meling, and Alberto Montresor. An Approach to Experimentally Obtain Service Dependability Characteristics of the Jgroup/ARM System. In *Proceedings of the Fifth European Dependable Computing Conference (EDCC)*, Lecture Notes in Computer Science, pages 179–198. Springer-Verlag, April 2005. pages 7, 195, 196, 215

[58] Matti A. Hiltunen and Richard D. Schlichting. The Cactus Approach to Building Configurable Middleware Services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nuremberg, Germany, October 2000. pages 74

[59] Second Workshop on Hot Topics in System Dependability. http://www.usenix.org/events/hotdep06/. Last visited May 2006. pages 233

[60] Norm C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991. pages 74

[61] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurements, Simulation, and Modeling*. John Wiley & Sons, Inc., 1991. pages 8

[62] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994. pages 14

[63] Kaustubh R. Joshi, Michel Cukier, and William H. Sanders. Experimental Evaluation of the Unavailability Induced by a Group Membership Protocol. In *Proceedings of the 4th European Dependable Computing Conference (EDCC)*, pages 140–158, Toulouse, France, October 2002. pages 200

[64] Christos T. Karamanolis and Jeff Magee. Client-access protocols for replicated services. *IEEE Transactions on Software Engineering*, 25(1), January 1999. pages 28, 29, 45, 57, 123, 124, 147, 197

[65] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. Moshe: A group membership service for WANs. *ACM Transactions on Computer Systems*, 20(3):191–238, August 2002. pages 112, 215

[66] Heine Kolltveit. High Availability Transactions. Master's thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, August 2005. pages 23, 36

[67] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990. pages 162

[68] Sacha Labourey and Bill Burke. *JBoss AS Clustering*. The JBoss Group, seventh edition, May 2004. pages 35

[69] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Internet Routing Instability. *IEEE/ACM Transactions on Networking*, 6(5):515–528, 1998. pages 215

[70] Leslise Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982. pages 15

[71] Ryan M. Lefever, Michel Cukier, and William H. Sanders. An Experimental Evaluation of Correlated Network Partitions in the Coda Distributed File System. In *Proceedings of the 22nd IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 273–282, Florence, Italy, October 2003. IEEE Computer Society. pages 174, 187, 215

[72] P. A. W. Lewis and E. J. Orav. *Simulation Methodology for Statisticians, Operation Analyst and Engineers*, volume 1 of *Statistics/Probability Series*. Wadsworth & Brooks/Cole, 1989. pages 196, 205, 210, 223, 225

[73] Silvano Maffeis. Adding Group Communication and Fault-Tolerance to CORBA. In *Proceedings of the 1st USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Monterey, CA, June 1995. pages 34

[74] Silvano Maffeis. The Object Group Design Pattern. In *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Toronto, Canada, June 1996. pages 27, 41

[75] Maven. http://maven.apache.org/. Last visited May 2006. pages 235

[76] Hein Meling and Bjarne E. Helvik. Dynamic Replication Management; Algorithm Specification. Plug-and-Play Technical Report 1/2000, Department of Telematics, Trondheim, Norway, December 2000. pages 32

[77] Hein Meling and Bjarne E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS)*, Bertinoro, Italy, May 2001. pages 55

[78] Hein Meling and Bjarne E. Helvik. Performance Consequences of Inconsistent Client-side Membership Information in the Open Group Model. In *Proceedings of the 23rd International Performance, Computing, and Communications Conference (IPCCC)*, Phoenix, Arizona, April 2004. pages 54, 123, 200

[79] Hein Meling, Jo Andreas Lind, and Henning Hommeland. Maintaining Binding Freshness in the Jgroup Dependable Naming Service. In *Proceedings of Norsk Informatikkonferanse (NIK)*, Oslo, Norway, November 2003. pages 129

[80] Hein Meling, Alberto Montresor, Özalp Babaoğlu, and Bjarne E. Helvik. Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing. Technical Report UBLCS-2002-12, Department of Computer Science, University of Bologna, October 2002. pages 39, 55, 195

[81] Hein Meling, Alberto Montresor, Bjarne E. Helvik, and Özalp Babaoğlu. Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management. Technical Report No. 11, University of Stavanger, January 2006. Submitted for publication. pages vii, 7, 34, 39, 55

[82] Sergio Mena, Xavier Cuvellier, Christophe Grégoire, and André Schiper. Appia vs. Cactus: Comparing Protocol Composition Frameworks. In *Proceedings of the 22nd IEEE International Symposium on Reliable Distributed Systems (SRDS)*, Florence, Italy, October 2003. pages 75

[83] David L. Mills. Network Time Protocol (Version 3); Specification, Implementation and Analysis, March 1992. RFC 1305. pages 178

[84] Hugo Miranda, Alexandre Pinto, and Luis Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, April 2001. pages 74, 76

[85] Rohnny Moland. Replicated Transactions in Jini: Integrating the Jini Transaction Service and Jgroup/ARM. Master's thesis, Department of Electrical and

Computer Engineering, Stavanger University College, June 2004. pages 23, 36

[86] Alberto Montresor. A Dependable Registry Service for the Jgroup Distributed Object Model. In *Proceedings of the 3rd European Research Seminar on Advances in Distributed Systems (ERSADS)*, Madeira, Portugal, April 1999. pages 45, 52, 54, 124, 128

[87] Alberto Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, Department of Computer Science, University of Bologna, February 2000. pages vii, 4, 5, 7, 9, 18, 20, 22, 26, 27, 29, 36, 39, 41, 42, 53, 54, 55, 78, 121, 195, 225

[88] Alberto Montresor, Renzo Davoli, and Özalp Babaoğlu. Enhancing Jini with Group Communication. In *Proceedings of the ICDCS Workshop on Applied Reliable Group Communication*, Phoenix, Arizona (USA), April 2001. Also appears as Technical Report UBLCS 2000-16, December 2000 (Revised January 2001). pages 23

[89] Alberto Montresor and Hein Meling. Jgroup Tutorial and Programmer's Manual. Technical Report UBLCS-2000-13, Department of Computer Science, University of Bologna, September 2000. Revised February 2002. pages 39

[90] Alberto Montresor, Hein Meling, and Özalp Babaoğlu. Messor: Load-Balancing through a Swarm of Autonomous Agents. In *Proceedings of the International Workshop on Agents and Peer-to-Peer Computing in conjunction with AAMAS 2002*, Bologna, Italy, July 2002. pages 7

[91] Graham Morgan, Santosh K. Shrivastava, Paul D. Ezhilchelvan, and Mark C. Little. Design and Implementation of a CORBA Fault-Tolerant Object Group Service. In *Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*, pages 361–374, Helsinki, Finland, June 1999. pages 34, 36

[92] Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Group Communication System. *Communications of the ACM*, 39(4), April 1996. pages 35

[93] Louise E. Moser, Peter M. Melliar-Smith, and Priya Narasimhan. Consistent Object Replication in the Eternal System. *Theory and Practice of Object Systems*, 4(2):81–92, January 1998. pages 4, 34, 35, 36, 37, 94

[94] Sape J. Mullender. Interprocess Communication. In Sape Mullender, editor, *Distributed Systems*, chapter 9, pages 217–250. Addison-Wesley, second edition, 1994. pages 147

[95] Richard Murch. *Autonomic Computing*. On Demand Series. IBM Press, 2004. pages 6, 56, 57

[96] Nitya Narasimhan. *Transparent Fault Tolerance for Java Remote Method Invocation*. PhD thesis, University of California, Santa Barbara, June 2001. pages 4, 34, 35, 36, 124

[97] Priya Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, December 1999. pages 34, 37, 195

[98] Priya Narasimhan, Louise E. Moser, and Peter M. Melliar-Smith. Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. In *Proceedings of the IEEE Symposium for Reliable Distributed Systems*, October 1999. Lausanne, Switzerland. pages 30

[99] Balachandran Natarajan, Aniruddha S. Gokhale, Shalini Yajnik, and Douglas C. Schmidt. DOORS: Towards High-performance Fault Tolerant CORBA. In *Proceedings of the 2nd International Symposium, Distributed Objects & Applications (DOA)*, pages 39–48, Antwerp, Belgium, September 2000. pages 37

[100] Soila Pertet and Priya Narasimhan. Proactive Recovery in Distributed CORBA Applications. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2004. pages 37

[101] Stefan Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, May 1994. pages 30

[102] David Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, pages 36–47, February 1994. pages 7, 17, 31, 36, 56, 145, 195, 197

[103] Yansong Ren. *AQuA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2001. pages 7, 37, 197

[104] Yansong Ren, David E. Bakken, Tod Courtney, Michel Cukier, David A. Karr, Paul Rubel, Chetan Sabnis, William H. Sanders, Richard E. Schantz, and Mouna Seri. AQuA: an adaptive architecture that provides dependable distributed objects. *IEEE Transactions on Computers*, 52(1):31–50, January 2003. pages 37, 56, 94, 145, 195, 197

[105] Carlos F. Reverte and Priya Narasimhan. Decentralized Resource Management and Fault-Tolerance for Distributed CORBA Applications. In *Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, 2003. pages 35, 37

[106] William H. Sanders and John F. Meyer. A Unified Approach for Specifying Measures of Performance, Dependability, and Performability. *Dependable Computing and Fault-Tolerant Systems: Dependable Computing for Critical Applications*, 4:215–237, 1991. pages 205

[107] Fred B. Schneider. What Good are Models and What Models are Good? In Sape Mullender, editor, *Distributed Systems*, chapter 2. Addison-Wesley, second edition, 1993. pages 30

[108] Fred B. Schneider. Replicated Management using the State-Machine Approach. In Sape Mullender, editor, *Distributed Systems*, chapter 7, pages 169–198. Addison-Wesley, second edition, 1994. pages 29, 162

[109] Secure shell. http://www.openssh.com/. Last visited May 2006. pages 66

[110] Mark E. Segal and Ophir Frieder. On-the-fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, pages 53–65, March 1993. pages 162

[111] D. P. Siewiorek and E. J. McCluskey. An Iterative Cell Switch Design. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, 1972. pages 32

[112] Kulpreet Singh. Towards Virtual Synchrony in MANETS, May 2005. Fifth European Dependable Computing Conference - Student Forum. pages 236

[113] Moris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4), 1994. pages 56, 137

[114] Marcin Solarski. *Dynamic Upgrade of Distributed Software Components*. PhD thesis, Technischen Universitat Berlin, January 2004. pages 57, 161, 162, 163, 168

[115] Marcin Solarski and Hein Meling. Towards Upgrading Actively Replicated Servers on-the-fly. In *Proceedings of the Workshop on Dependable On-line Upgrading of Distributed Systems in conjunction with COMPSAC 2002*, Oxford, England, August 2002. pages 57, 161, 162, 167, 168

[116] Frank Sommers. Call on extensible RMI: An introduction to JERI. http://www.javaworld.com/javaworld/jw-12-2003/jw-1219-jiniology_p.html, December 2003. Last visited May 2006. pages 24

[117] The Spread Toolkit. http://www.spread.org/. Last visited May 2006. pages 4

[118] Tor Arve Stangeland. Client Multicast in Jgroup. Master's thesis, Department of Electrical and Computer Engineering, Stavanger University College, June 2003. pages 30, 97

[119] David T. Stott, Benjamin Floering, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proceedings of the 4th International Computer Performance and Dependability Symposium*, 2000. pages 174

[120] Sun Microsystems, Santa Clara, CA. *Enterprise JavaBeans Specification, Version 2.1*, November 2003. pages 4, 18, 25, 36

[121] Sun Microsystems, Santa Clara, CA. *Jini Architecture Specification, Version 2.0*, June 2003. pages 23

[122] Sun Microsystems, Santa Clara, CA. *Jini Technology Core Platform Specification, Version 2.0*, June 2003. pages 23

[123] Sun Microsystems, Santa Clara, CA. *Java Remote Method Invocation Specification, Rev. 1.10*, February 2004. pages 4, 18, 69

[124] Diana Szentivanyi. *Performance Studies of Fault-Tolerant Middleware*. PhD thesis, Linköpings universitet, 2005. pages 34, 35

[125] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems – Principles and Paradigms*. Prentice Hall, 2002. pages 4, 5, 18, 40, 95

[126] The TAPAS project (formerly Plug-and-Play). http://tapas.item.ntnu.no/. Last visited May 2006. pages 6

[127] Lauren A. Tewksbury, Louise E. Moser, and Peter M. Melliar-Smith. Live Upgrade Techniques for CORBA Applications. In *Proceedings of the 3rd Int'l Working Conference on Distributed Applications and Interoperable Systems*, Krakow, Poland, September 2001. pages 162

[128] The TINA Consortium. http://www.tinac.com/. Last visited May 2006. pages 4

[129] TINA Consortium. *TINA-C Deliverable: Overall Concepts and Principles of TINA, V1.0*, February 1995. pages 4

[130] TINA Consortium. *TINA-C Deliverable: Service Architecture, V5.0*, June 1997. pages 4

[131] Robbert van Renesse. Masking the Overhead of Layering. In *Proceedings of the 1996 ACM SIGCOMM Conference*, Stanford University, August 1996. pages 74

[132] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996. pages 26, 28, 74, 76

[133] Rune Vestvik. Pålitelighetsvurdering og integrasjonstesting av distribuerte applikasjoner. Master's thesis, Department of Electrical and Computer Engineering, University of Stavanger, June 2005. In Norwegian. pages 177

[134] Matthias Wiesmann. *Group Communications and Database Replication: Techniques, Issues and Performance*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2002. pages 5

[135] Otto Wittner. *Emergent Behavior Based Implements for Distributed Network Management*. PhD thesis, Department of Telematics, Norwegian University of Science and Technology, November 2003. pages 7

[136] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Toronto, Canada, June 1996. pages 20